O'REILLY

TURING 图灵程序设计丛书



Java 8 函数式编程

Java 8 Lambdas: Functional Programming for the Masses

[英] Richard Warburton 著

王群锋 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍



王群锋

毕业于西安电子科技大学，现任职于IBM西安研发中心，从事下一代统计预测软件的开发运维工作。

TURING 图灵程序设计丛书

Java 8函数式编程

Java 8 Lambdas
Functional Programming For The Masses

［英］Richard Warburton 著
王群锋 译

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

Java 8函数式编程 / (英) 沃伯顿 (Warburton,R.)
著 ; 王群锋译. -- 北京 : 人民邮电出版社, 2015.4
（图灵程序设计丛书）
ISBN 978-7-115-38488-1

Ⅰ. ①J… Ⅱ. ①沃… ②王… Ⅲ. ①JAVA语言—程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第025390号

内容提要

多年以来，函数式编程被认为是少数人的游戏，不适合推广给普罗大众。写作此书的目的就是为了挑战这种思想。本书将探讨如何编写出简单、干净、易读的代码；如何简单地使用并行计算提高性能；如何准确地为问题建模，并且开发出更好的领域特定语言；如何写出不易出错，并且更简单的并发代码；如何测试和调试 Lambda 表达式。

如果你已经掌握 Java SE，想尽快了解 Java 8 新特性，写出简单干净的代码，那么本书不容错过。

◆ 著　[英] Richard Warburton
译　王群锋
责任编辑　李松峰
执行编辑　李　静　仇祝平
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：9.25
字数：191千字　2015年 4 月第 1 版
印数：1－3 500册　2015年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-6949号

定价：39.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

多年以来，函数式编程被认为是少数人的游戏，这些人总是强调自己在智力上的优越性，认为函数式编程的智慧不适合推广给普罗大众。写作此书的目的就是为了挑战这种思想，函数式编程并没有多么了不起，也绝不是少数人的游戏。

在过去的两年中，我请伦敦 Java 社区的开发人员以各种方式测试 Java 8 的新特性。我发现很多人都喜欢 Java 8 的新用法和类库。他们有可能被一些术语和高大上的概念吓到，但是稍稍一丁点儿函数式编程技巧都能给编程带来便利，他们对此喜不自胜。人们津津乐道的话题之一是使用新的 Stream API 操作对象和集合类时（比如从所有的唱片列表中过滤出在英国本地出品的唱片时），代码是多么易读。

组织这些 Java 社区活动，让我认识到了示例代码的重要性。人们通过不断地阅读和消化这些简单的示例，最终归纳出某种模式。我还意识到术语是多么令人讨厌，因此，在介绍一个晦涩的概念时，我都会给出通俗易懂的解释。

对很多人来说，Java 8 提供的函数式编程元素有限：没有单子[1]，没有语言层面的惰性求值，也没有为不可变性提供额外支持。对实用至上的程序员来说，这没什么大不了的，我们只想在类库级别抽象，写出简单干净的代码来解决业务问题。如果有人为我们写出这样的类库，那再好不过了，这样我们就可以把主要精力放在日常工作上了。

为什么要阅读本书

本书将探讨如下主题。

- 如何编写出简单、干净、易读的代码——尤其是对于集合的操作？
- 如何简单地使用并行计算提高性能？

注 1：别担心，这是本书唯一提及单子的地方。

- 如何准确地为问题建模，并且开发出更好的领域特定语言？
- 如何写出不易出错，并且更简单的并发代码？
- 如何测试和调试 Lambda 表达式？

将 Lambda 表达式加入 Java，并不只是为了提高开发人员的生产效率，业界也对这一特性有根本性的需求。

本书读者对象

本书面向那些已经掌握 Java SE，并且想尽快了解 Java 8 新特性的开发人员。

如果你对 Lambda 表达式感兴趣，想知道它怎么帮助你提升专业技能，那么这本书就是为你而写的。我假设读者还不知道 Lambda 表达式，以及 Java 8 中核心类库的变化，我将从零开始介绍这些概念、类库和技术。

虽然我想让所有开发人员都来买这本书，但这不现实，这不是一本适合所有人的书。如果你一点儿也不懂 Java，那么这本书就不适合你。同时，尽管本书会详细讲解 Java 中的 Lambda 表达式，但是我不会解释怎样在其他语言中使用 Lambda 表达式。

我也不会讲解 Java SE 中一些基本的概念，比如集合类、匿名内部类或者 Swing 中的事件处理机制。我假设读者已经掌握了这些知识。

怎样阅读本书

本书采用了示例驱动的写作风格：介绍完一个概念之后，就会紧跟一段代码。代码中的一些片段，有时你可能无法全部看懂。没关系，通常在代码后面会紧跟一段文字，讲解代码的细节。

这种方式能让你边学边练，多数章节还在最后提供了练习题，供读者自行练习。我强烈建议读者读完一章后完成这些练习，熟能生巧。每个务实的程序员都知道，自欺欺人很容易，你觉得读懂一段代码了，其实还是遗漏了一些细节。

使用 Lambda 表达式，就是将复杂性抽象到类库的过程。在本书中，我会引入很多常用类库的细节。第 2 章至第 6 章介绍了 JDK 8 中核心语言的变化以及升级后的类库。

最后三章介绍了如何在真实环境下使用函数式编程。第 7 章介绍一些让测试和调试 Lambda 表达式变得容易的技巧；第 8 章解释现有的那些良好的软件设计原则如何应用到 Lambda 表达式上；第 9 章讨论并发，怎样使用 Lambda 表达式写出易读且易于维护的并发代码。涉及第三方类库时，这些章节也会加以介绍。

读者可以将前四章当作 Java 8 的入门指南——要用好 Java 8，每个人都必须学会这些知识。

后面的几章难度略高，但掌握了这几章的内容，你就可以成为知识更加全面的程序员，在自己的设计中得心应手地使用 Lambda 表达式。你在不断学习的过程中，也会接触大量的练习，答案可以在 GitHub（https://github.com/RichardWarburton/java-8-Lambdas-exercises）上找到。如果你能边学边练，就能迅速掌握 Lambda 表达式。

本书排版规范

本书使用以下排版规范。

- 楷体

  表示新术语。

- 等宽字体

  表示程序片段，也用于在正文中表示程序中使用的变量、函数名、数据库、数据类型、环境变量、语句和关键字等元素。

- 等宽粗体

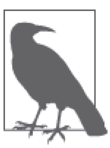
  表示应该由用户逐字输入的命令或者其他文本。

- 等宽斜体

  表示将由用户提供的值（或由上下文确定的值）替换的文本。



这个图标表示提示或建议。



这个图标表示重要说明。



这个图标表示警告或提醒。

使用代码示例

可以在这里下载本书随附的资料（代码示例、练习题等）：https://github.com/RichardWarburton/java-8-lambdas-exercises。

让本书助你一臂之力。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布 O'Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处，但不强求。出处信息一般包括书名、作者、出版商和书号，例如：*Java 8 Lambdas*，Richard Warburton 著（O'Reilly，2014）。版权所有，978-1-449-37077-0。

如果还有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/java_8_lambdas。

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

虽然本书的封面上署的是我的名字，但本书得以出版要归功于很多人。

首先要感谢我的编辑 Meghan 和 O'Reilly 的出版团队，他们让整个出版过程变得很愉快，而且他们还适当加快了本书的出版进度。还要感谢 Martijn 和 Ben 将我引荐给 Meghan，没有这次会面就不会有这本书。

审阅过程极大地提升了本书的质量，衷心感谢那些正式或非正式参与审阅的朋友，他们是：Martijn Verburg、Jim Gough、John Oliver、Edward Wong、Brian Goetz、Daniel Bryant、Fred Rosenberger、Jaikiran Pai 和 Mani Sarkar。尤其要感谢 Martijn，他给了我如何写一本技术书的实战指导。

如果忘记感谢 Oracle 公司的 Project Lambda 项目组，我不会原谅自己。更新一个成熟的语言是一项巨大的挑战，他们不辱使命，我也因此有了得以编写本书的素材。在 Java 8 发布早期版本时，伦敦的 Java 社区积极参与测试，通过这些测试，很容易就发现了开发人员犯了哪类错误，哪些地方可以修复，感谢他们！

在写作本书的过程中，我得到了很多人的支持和帮助，特别是我的父母。在我需要的时候，他们总是陪伴在身边。我的朋友们也总是给我积极的评价和鼓励，包括 Compsoc 里的那些老伙计们，特别是 Sadiq Jaffer 和基督少年军，感谢你们！

第 1 章 简介

在开始探索 Lambda 表达式之前，首先我们要知道它因何而生。本章将介绍 Lambda 表达式产生的原因，以及本书的写作动机和组织结构。

1.1 为什么需要再次修改Java

1996 年 1 月，Java 1.0 发布，此后计算机编程领域发生了翻天覆地的变化。商业发展需要更复杂的应用，大多数程序都跑在功能强大的多核 CPU 的机器上。带有高效运行时编译器的 Java 虚拟机（JVM）的出现，使程序员将更多精力放在编写干净、易于维护的代码上，而不是思考如何将每一个 CPU 时钟周期、每字节内存物尽其用。

多核 CPU 的兴起成为了不容回避的事实。涉及锁的编程算法不但容易出错，而且耗费时间。人们开发了 `java.util.concurrent` 包和很多第三方类库，试图将并发抽象化，帮助程序员写出在多核 CPU 上运行良好的程序。很可惜，到目前为止，我们的成果还远远不够。

开发类库的程序员使用 Java 时，发现抽象级别还不够。处理大型数据集合就是个很好的例子，面对大型数据集合，Java 还欠缺高效的并行操作。开发者能够使用 Java 8 编写复杂的集合处理算法，只需要简单修改一个方法，就能让代码在多核 CPU 上高效运行。为了编写这类处理批量数据的并行类库，需要在语言层面上修改现有的 Java：增加 Lambda 表达式。

当然，这样做是有代价的，程序员必须学习如何编写和阅读使用 Lambda 表达式的代码，但是，这不是一桩赔本的买卖。与手写一大段复杂、线程安全的代码相比，学习一点新语法和一些新习惯容易很多。开发企业级应用时，好的类库和框架极大地降低了开发时间和成本，也为开发易用且高效的类库扫清了障碍。

对于习惯了面向对象编程的开发者来说，抽象的概念并不陌生。面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因此这两种编程方式我们都得学。

这种新的抽象方式还有其他好处。不是所有人都在编写性能优先的代码，对于这些人来说，函数式编程带来的好处尤为明显。程序员能编写出更容易阅读的代码——这种代码更多地表达了业务逻辑的意图，而不是它的实现机制。易读的代码也易于维护、更可靠、更不容易出错。

在写回调函数和事件处理程序时，程序员不必再纠缠于匿名内部类的冗繁和可读性，函数式编程让事件处理系统变得更加简单。能将函数方便地传递也让编写惰性代码变得容易，惰性代码在真正需要时才初始化变量的值。

Java 8 还让集合类可以拥有一些额外的方法：`default` 方法。程序员在维护自己的类库时，可以使用这些方法。

总而言之，Java 已经不是祖辈们当年使用的 Java 了，嗯，这不是件坏事。

1.2　什么是函数式编程

每个人对*函数式编程*的理解不尽相同。但其核心是：在思考问题时，使用不可变值和函数，函数对一个值进行处理，映射成另一个值。

不同的语言社区往往对各自语言中的特性孤芳自赏。现在谈 Java 程序员如何定义函数式编程还为时尚早，但是，这根本不重要！我们关心的是如何写出好代码，而不是符合函数式编程风格的代码。

本书将重点放在函数式编程的实用性上，包括可以被大多数程序员理解和使用的技术，帮助他们写出易读、易维护的代码。

1.3　示例

本书中的示例全部都围绕一个常见的问题领域构造：音乐。具体来说，这些示例代表了在专辑上常常看到的信息，有关术语定义如下。

- `Artist`

  创作音乐的个人或团队。

  - `name`：艺术家的名字（例如“甲壳虫乐队”）。
  - `members`：乐队成员（例如“约翰 · 列侬”），该字段可为空。
  - `origin`：乐队来自哪里（例如“利物浦”）。

- `Track`

  专辑中的一支曲目。

  - `name`：曲目名称（例如《黄色潜水艇》）。

- `Album`

  专辑，由若干曲目组成。

  - `name`：专辑名（例如《左轮手枪》）。
  - `tracks`：专辑上所有曲目的列表。
  - `musicians`：参与创作本专辑的艺术家列表。

本书将使用这个问题讲解如何在正常的业务领域或者 Java 应用中使用函数式编程技术。也许读者认为这些示例并不完美，但它和真实的业务领域应用比起来足够简单，书中的很多代码都是基于这个简单的模型。

第 2 章

Lambda表达式

Java 8 的最大变化是引入了 Lambda 表达式——一种紧凑的、传递行为的方式。它也是本书后续章节所述内容的基础，因此，接下来就了解一下什么是 Lambda 表达式。

2.1 第一个Lambda表达式

Swing 是一个与平台无关的 Java 类库，用来编写图形用户界面（GUI）。该类库有一个常见用法：为了响应用户操作，需要注册一个事件监听器。用户一输入，监听器就会执行一些操作（见例 2-1）。

例 2-1　使用匿名内部类将行为和按钮单击进行关联

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

在这个例子中，我们创建了一个新对象，它实现了 `ActionListener` 接口。这个接口只有一个方法 `actionPerformed`，当用户点击屏幕上的按钮时，`button` 就会调用这个方法。匿名内部类实现了该方法。在例 2-1 中该方法所执行的只是输出一条信息，表明按钮已被点击。

> 这实际上是一个代码即数据的例子——我们给按钮传递了一个代表某种行为的对象。

设计匿名内部类的目的，就是为了方便 Java 程序员将代码作为数据传递。不过，匿名内部类还是不够简便。为了调用一行重要的逻辑代码，不得不加上 4 行冗繁的样板代码。若把样板代码用其他颜色区分开来，就可一目了然：

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

尽管如此，样板代码并不是唯一的问题：这些代码还相当难读，因为它没有清楚地表达程序员的意图。我们不想传入对象，只想传入行为。在 Java 8 中，上述代码可以写成一个 Lambda 表达式，如例 2-2 所示。

例 2-2　使用 Lambda 表达式将行为和按钮单击进行关联

```
button.addActionListener(event -> System.out.println("button clicked"));
```

和传入一个实现某接口的对象不同，我们传入了一段代码块——一个没有名字的函数。`event` 是参数名，和上面匿名内部类示例中的是同一个参数。`->` 将参数和 Lambda 表达式的主体分开，而主体是用户点击按钮时会运行的一些代码。

和使用匿名内部类的另一处不同在于声明 `event` 参数的方式。使用匿名内部类时需要显式地声明参数类型 `ActionEvent event`，而在 Lambda 表达式中无需指定类型，程序依然可以编译。这是因为 `javac` 根据程序的上下文（`addActionListener` 方法的签名）在后台推断出了参数 `event` 的类型。这意味着如果参数类型不言而明，则无需显式指定。稍后会介绍类型推断的更多细节，现在先来看看编写 Lambda 表达式的各种方式。

> 尽管与之前相比，Lambda 表达式中的参数需要的样板代码很少，但是 Java 8 仍然是一种静态类型语言。为了增加可读性并迁就我们的习惯，声明参数时也可以包括类型信息，而且有时编译器不一定能根据上下文推断出参数的类型！

2.2　如何辨别Lambda表达式

Lambda 表达式除了基本的形式之外，还有几种变体，如例 2-3 所示。

例 2-3　编写 Lambda 表达式的不同形式

```
Runnable noArguments = () -> System.out.println("Hello World"); ❶

ActionListener oneArgument = event -> System.out.println("button clicked"); ❷

Runnable multiStatement = () -> { ❸
```

```
        System.out.print("Hello");
        System.out.println(" World");
    };

    BinaryOperator<Long> add = (x, y) -> x + y;  ❹

    BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

❶中所示的 Lambda 表达式不包含参数，使用空括号 () 表示没有参数。该 Lambda 表达式实现了 `Runnable` 接口，该接口也只有一个 `run` 方法，没有参数，且返回类型为 `void`。

❷中所示的 Lambda 表达式包含且只包含一个参数，可省略参数的括号，这和例 2-2 中的形式一样。

Lambda 表达式的主体不仅可以是一个表达式，而且也可以是一段代码块，使用大括号（{}）将代码块括起来，如❸所示。该代码块和普通方法遵循的规则别无二致，可以用返回或抛出异常来退出。只有一行代码的 Lambda 表达式也可使用大括号，用以明确 Lambda 表达式从何处开始、到哪里结束。

Lambda 表达式也可以表示包含多个参数的方法，如❹所示。这时就有必要思考怎样去阅读该 Lambda 表达式。这行代码并不是将两个数字相加，而是创建了一个函数，用来计算两个数字相加的结果。变量 `add` 的类型是 `BinaryOperator<Long>`，它不是两个数字的和，而是将两个数字相加的那行代码。

到目前为止，所有 Lambda 表达式中的参数类型都是由编译器推断得出的。这当然不错，但有时最好也可以显式声明参数类型，此时就需要使用小括号将参数括起来，多个参数的情况也是如此。如❺所示。

> *目标类型*是指 Lambda 表达式所在上下文环境的类型。比如，将 Lambda 表达式赋值给一个局部变量，或传递给一个方法作为参数，局部变量或方法参数的类型就是 Lambda 表达式的目标类型。

上述例子还隐含了另外一层意思：Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的。目标类型也不是一个全新的概念。如例 2-4 所示，Java 中初始化数组时，数组的类型就是根据上下文推断出来的。另一个常见的例子是 `null`，只有将 `null` 赋值给一个变量，才能知道它的类型。

例 2-4　等号右边的代码并没有声明类型，系统根据上下文推断出类型信息

```
final String[] array = { "hello", "world" };
```

2.3 引用值，而不是变量

如果你曾使用过匿名内部类，也许遇到过这样的情况：需要引用它所在方法里的变量。这时，需要将变量声明为 `final`，如例 2-5 所示。将变量声明为 `final`，意味着不能为其重复赋值。同时也意味着在使用 `final` 变量时，实际上是在使用赋给该变量的一个特定的值。

例 2-5　匿名内部类中使用 `final` 局部变量

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("hi " + name);
    }
});
```

Java 8 虽然放松了这一限制，可以引用非 `final` 变量，但是该变量在既成事实上必须是 `final`。虽然无需将变量声明为 `final`，但在 Lambda 表达式中，也无法用作非终态变量。如果坚持用作非终态变量，编译器就会报错。

既成事实上的 `final` 是指只能给该变量赋值一次。换句话说，Lambda 表达式引用的是值，而不是变量。在例 2-6 中，`name` 就是一个既成事实上的 `final` 变量。

例 2-6　Lambda 表达式中引用既成事实上的 `final` 变量

```
String name = getUserName();
button.addActionListener(event -> System.out.println("hi " + name));
```

`final` 就像代码中的线路噪声，省去之后代码更易读。当然，有些情况下，显式地使用 `final` 代码更易懂。是否使用这种既成事实上的 `final` 变量，完全取决于个人喜好。

如果你试图给该变量多次赋值，然后在 Lambda 表达式中引用它，编译器就会报错。比如，例 2-7 无法通过编译，并显示出错信息：`local variables referenced from a Lambda expression must be final or effectively final`[1]。

例 2-7　未使用既成事实上的 `final` 变量，导致无法通过编译

```
String name = getUserName();
name = formatUserName(name);
button.addActionListener(event -> System.out.println("hi " + name));
```

这种行为也解释了为什么 Lambda 表达式也被称为闭包。未赋值的变量与周边环境隔离起来，进而被绑定到一个特定的值。在众说纷纭的计算机编程语言圈子里，Java 是否拥有真正的闭包一直备受争议，因为在 Java 中只能引用既成事实上的 `final` 变量。*名字虽异，功能相同*，就好比把菠萝叫作凤梨，其实都是同一种水果。为了避免无意义的争论，全书将使用“Lambda 表达式”一词。无论名字如何，如前文所述，Lambda 表达式都是静态类型

注 1：Lambda 表达式中引用的局部变量必须是 `final` 或既成事实上的 `final` 变量。——译者注

的。因此，接下来就分析一下 Lambda 表达式本身的类型：*函数接口*。

2.4　函数接口

> 函数接口是只有一个抽象方法的接口，用作 Lambda 表达式的类型。

在 Java 里，所有方法参数都有固定的类型。假设将数字 `3` 作为参数传给一个方法，则参数的类型是 `int`。那么，Lambda 表达式的类型又是什么呢？

使用只有一个方法的接口来表示某特定方法并反复使用，是很早就有的习惯。使用 Swing 编写过用户界面的人对这种方式都不陌生，例 2-2 中的用法也是如此。这里无需再标新立异，Lambda 表达式也使用同样的技巧，并将这种接口称为*函数接口*。例 2-8 展示了前面例子中所用的函数接口。

例 2-8　`ActionListener` 接口：接受 `ActionEvent` 类型的参数，返回空

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent event);
}
```

`ActionListener` 只有一个抽象方法：`actionPerformed`，被用来表示行为：接受一个参数，返回空。记住，由于 `actionPerformed` 定义在一个接口里，因此 `abstract` 关键字不是必需的。该接口也继承自一个不具有任何方法的父接口：`EventListener`。

这就是函数接口，接口中单一方法的命名并不重要，只要方法签名和 Lambda 表达式的类型匹配即可。可在函数接口中为参数起一个有意义的名字，增加代码易读性，便于更透彻地理解参数的用途。

这里的函数接口接受一个 `ActionEvent` 类型的参数，返回空（`void`），但函数接口还可有其他形式。例如，函数接口可以接受两个参数，并返回一个值，还可以使用泛型，这完全取决于你要干什么。

以后我将使用图形来表示不同类型的函数接口。指向函数接口的箭头表示参数，如果箭头从函数接口射出，则表示方法的返回类型。`ActionListener` 的函数接口如图 2-1 所示。

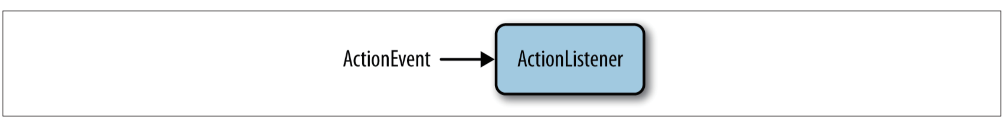


图 2-1：`ActionListener` 接口，接受一个 `ActionEvent` 对象，返回空

使用 Java 编程，总会遇到很多函数接口，但 Java 开发工具包（JDK）提供的一组核心函数接口会频繁出现。表 2-1 罗列了一些最重要的函数接口。

表2-1　Java中重要的函数接口

接口	参数	返回类型	示例
Predicate<T>	T	boolean	这张唱片已经发行了吗
Consumer<T>	T	void	输出一个值
Function<T,R>	T	R	获得 Artist 对象的名字
Supplier<T>	None	T	工厂方法
UnaryOperator<T>	T	T	逻辑非（!）
BinaryOperator<T>	(T, T)	T	求两个数的乘积（*）

前面已讲过函数接口接收的类型，也讲过 javac 可以根据上下文自动推断出参数的类型，且用户也可以手动声明参数类型，但何时需要手动声明呢？下面将对类型推断作详尽说明。

2.5　类型推断

某些情况下，用户需要手动指明类型，建议大家根据自己或项目组的习惯，采用让代码最便于阅读的方法。有时省略类型信息可以减少干扰，更易弄清状况；而有时却需要类型信息帮助理解代码。经验证发现，一开始类型信息是有用的，但随后可以只在真正需要时才加上类型信息。下面将介绍一些简单的规则，来帮助确认是否需要手动声明参数类型。

Lambda 表达式中的类型推断，实际上是 Java 7 就引入的目标类型推断的扩展。读者可能已经知道 Java 7 中的菱形操作符，它可使 javac 推断出泛型参数的类型。参见例 2-9。

例 2-9　使用菱形操作符，根据变量类型做推断

```
Map<String, Integer> oldWordCounts = new HashMap<String, Integer>();  ❶
Map<String, Integer> diamondWordCounts = new HashMap<>();  ❷
```

我们为变量 oldWordCounts ❶明确指定了泛型的类型，而变量 diamondWordCounts ❷则使用了菱形操作符。不用明确声明泛型类型，编译器就可以自己推断出来，这就是它的神奇之处！

当然，这并不是什么魔法，根据变量 diamondWordCounts ❷的类型可以推断出 HashMap 的泛型类型，但用户仍需要声明变量的泛型类型。

如果将构造函数直接传递给一个方法，也可根据方法签名来推断类型。在例 2-10 中，我们传入了 HashMap，根据方法签名已经可以推断出泛型的类型。

例 2-10　使用菱形操作符，根据方法签名做推断

```
useHashmap(new HashMap<>());

...
```

```java
private void useHashmap(Map<String, String> values);
```

Java 7 中程序员可省略构造函数的泛型类型，Java 8 更进一步，程序员可省略 Lambda 表达式中的所有参数类型。再强调一次，这并不是魔法，`javac` 根据 Lambda 表达式上下文信息就能推断出参数的正确类型。程序依然要经过类型检查来保证运行的安全性，但不用再显式声明类型罢了。这就是所谓的*类型推断*。

> Java 8 中对类型推断系统的改善值得一提。上面的例子将 `new HashMap<>()` 传给 `useHashmap` 方法，即使编译器拥有足够的信息，也无法在 Java 7 中通过编译。

接下来将通过举例来详细分析类型推断。

例 2-11 和例 2-12 都将变量赋给一个函数接口，这样便于理解。第一个例子（例 2-11）使用 Lambda 表达式检测一个 `Integer` 是否大于 `5`。这实际上是一个 `Predicate`——用来判断真假的函数接口。

例 2-11　类型推断

```java
Predicate<Integer> atLeast5 = x -> x > 5;
```

`Predicate` 也是一个 Lambda 表达式，和前文中 `ActionListener` 不同的是，它还返回一个值。在例 2-11 中，表达式 `x > 5` 是 Lambda 表达式的主体。这样的情况下，返回值就是 Lambda 表达式主体的值。

例 2-12　`Predicate` 接口的源码，接受一个对象，返回一个布尔值

```java
public interface Predicate<T> {
    boolean test(T t);
}
```

从例 2-12 中可以看出，`Predicate` 只有一个泛型类型的参数，`Integer` 用于其中。Lambda 表达式实现了 `Predicate` 接口，因此它的单一参数被推断为 `Integer` 类型。`javac` 还可检查 Lambda 表达式的返回值是不是 `boolean`，这正是 `Predicate` 方法的返回类型（如图 2-2）。

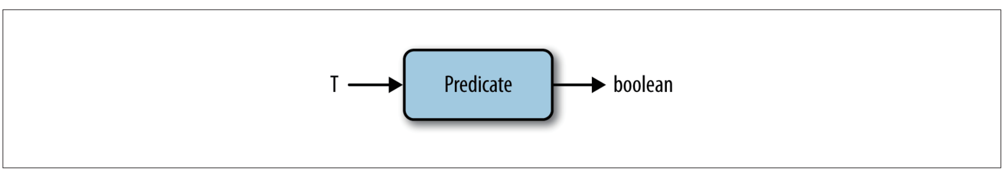


图 2-2：`Predicate` 接口图示，接受一个对象，返回一个布尔值

例 2-13 是一个略显复杂的函数接口：`BinaryOperator`。该接口接受两个参数，返回一个

值，参数和值的类型均相同。实例中所用的类型是 Long。

例 2-13　略显复杂的类型推断

```
BinaryOperator<Long> addLongs = (x, y) -> x + y;
```

类型推断系统相当智能，但若信息不够，类型推断系统也无能为力。类型系统不会漫无边际地瞎猜，而会中止操作并报告编译错误，寻求帮助。比如，如果我们删掉例 2-13 中的某些类型信息，就会得到例 2-14 所示的代码。

例 2-14　没有泛型，代码则通不过编译

```
BinaryOperator add = (x, y) -> x + y;
```

编译器给出的报错信息如下：

```
Operator '& #x002B;' cannot be applied to java.lang.Object, java.lang.Object.
```

报错信息让人一头雾水，到底怎么回事？ BinaryOperator 毕竟是一个具有泛型参数的函数接口，该类型既是参数 x 和 y 的类型，也是返回值的类型。上面的例子中并没有给出变量 add 的任何泛型信息，给出的正是原始类型的定义。因此，编译器认为参数和返回值都是 java.lang.Object 实例。

4.3 节还会讲到类型推断，但就目前来说，掌握以上类型推断的知识就已经足够了。

2.6　要点回顾

- Lambda 表达式是一个匿名方法，将行为像数据一样进行传递。
- Lambda 表达式的常见结构：BinaryOperator<Integer> add = (x, y) → x + y。
- 函数接口指仅具有单个抽象方法的接口，用来表示 Lambda 表达式的类型。

2.7　练习

每章最后都附有一组练习，帮助读者实践并巩固本章的知识和新概念。练习答案可在 GitHub（https://github.com/RichardWarburton/java-8-Lambdas-exercises）上本书所对应的代码仓库中找到。

1. 请看例 2-15 中的 Function 函数接口并回答下列问题。

例 2-15　Function 函数接口

```
public interface Function<T, R> {
    R apply(T t);
}
```

a. 请画出该函数接口的图示。

b. 若要编写一个计算器程序，你会使用该接口表示什么样的 Lambda 表达式？

c. 下列哪些 Lambda 表达式有效实现了 `Function<Long,Long>`？

```
x -> x + 1;
(x, y) -> x + 1;
x -> x == 1;
```

2. ThreadLocal Lambda 表达式。Java 有一个 `ThreadLocal` 类，作为容器保存了当前线程里局部变量的值。Java 8 为该类新加了一个工厂方法，接受一个 Lambda 表达式，并产生一个新的 `ThreadLocal` 对象，而不用使用继承，语法上更加简洁。

a. 在 Javadoc 或集成开发环境（IDE）里找出该方法。

b. `DateFormatter` 类是非线程安全的。使用构造函数创建一个线程安全的 `DateFormatter` 对象，并输出日期，如“01-Jan-1970”。

3. 类型推断规则。下面是将 Lambda 表达式作为参数传递给函数的一些例子。`javac` 能正确推断出 Lambda 表达式中参数的类型吗？换句话说，程序能编译吗？

a. `Runnable helloWorld = () -> System.out.println("hello world");`

b. 使用 Lambda 表达式实现 `ActionListener` 接口：

```
JButton button = new JButton();
button.addActionListener(event ->
    System.out.println(event.getActionCommand()));
```

c. 以如下方式重载 `check` 方法后，还能正确推断出 `check(x -> x > 5)` 的类型吗？

```
interface IntPred {
    boolean test(Integer value);
}
boolean check(Predicate<Integer> predicate);

boolean check(IntPred predicate);
```

你可能需要查阅 Javadoc 或在 IDE 里查看方法的参数类型，验证重载是否有效。

第 3 章

流

Java 8 中新增的特性旨在帮助程序员写出更好的代码，其中对核心类库的改进是很关键的一部分，也是本章的主要内容。对核心类库的改进主要包括集合类的 API 和新引入的流（Stream）。流使程序员得以站在更高的抽象层次上对集合进行操作。

本章会介绍 `Stream` 类中的一组方法，每个方法都对应集合上的一种操作。

3.1　从外部迭代到内部迭代

> 本章及本书其余部分的例子大多围绕 1.3 节介绍的案例展开。

Java 程序员在使用集合类时，一个通用的模式是在集合上进行迭代，然后处理返回的每一个元素。比如要计算从伦敦来的艺术家的人数，通常代码会写成例 3-1 这样。

例 3-1　使用 `for` 循环计算来自伦敦的艺术家人数

```
int count = 0;
for (Artist artist : allArtists) {
    if (artist.isFrom("London")) {
        count++;
    }
}
```

尽管这样的操作可行，但存在几个问题。每次迭代集合类时，都需要写很多样板代码。将

for 循环改造成并行方式运行也很麻烦，需要修改每个 for 循环才能实现。

此外，上述代码无法流畅传达程序员的意图。for 循环的样板代码模糊了代码的本意，程序员必须阅读整个循环体才能理解。若是单一的 for 循环，倒也问题不大，但面对一个满是循环（尤其是嵌套循环）的庞大代码库时，负担就重了。

就其背后的原理来看，for 循环其实是一个封装了迭代的语法糖，我们在这里多花点时间，看看它的工作原理。首先调用 iterator 方法，产生一个新的 Iterator 对象，进而控制整个迭代过程，这就是外部迭代。迭代过程通过显式调用 Iterator 对象的 hasNext 和 next 方法完成迭代。展开后的代码如例 3-2 所示，图 3-1 展示了迭代过程中的方法调用。

例 3-2　使用迭代器计算来自伦敦的艺术家人数

```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

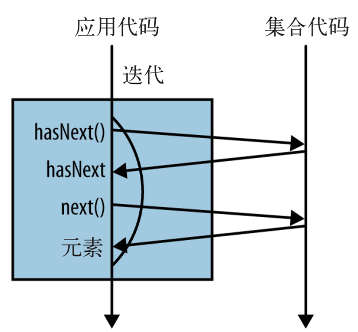


图 3-1：外部迭代

然而，外部迭代也有问题。首先，它很难抽象出本章稍后提及的不同操作；此外，它从本质上来讲是一种串行化操作。总体来看，使用 for 循环会将行为和方法混为一谈。

另一种方法就是内部迭代，如例 3-3 所示。首先要注意 stream() 方法的调用，它和例 3-2 中调用 iterator() 的作用一样。该方法不是返回一个控制迭代的 Iterator 对象，而是返回内部迭代中的相应接口：Stream。

例 3-3　使用内部迭代计算来自伦敦的艺术家人数

```
long count = allArtists.stream()
                       .filter(artist -> artist.isFrom("London"))
                       .count();
```

图 3-2 展示了使用类库后的方法调用流程，与图 3-1 形成对比。

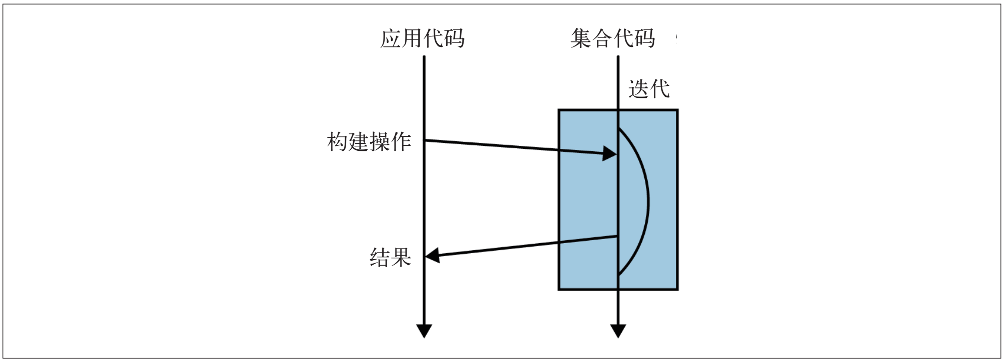
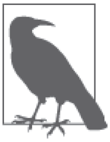


图 3-2：内部迭代



`Stream` 是用函数式编程方式在集合类上进行复杂操作的工具。

例 3-3 可被分解为两步更简单的操作：

- 找出所有来自伦敦的艺术家；
- 计算他们的人数。

每种操作都对应 `Stream` 接口的一个方法。为了找出来自伦敦的艺术家，需要对 `Stream` 对象进行过滤：`filter`。过滤在这里是指“只保留通过某项测试的对象”。测试由一个函数完成，根据艺术家是否来自伦敦，该函数返回 `true` 或者 `false`。由于 Stream API 的函数式编程风格，我们并没有改变集合的内容，而是描述出 `Stream` 里的内容。`count()` 方法计算给定 `Stream` 里包含多少个对象。

3.2 实现机制

例 3-3 中，整个过程被分解为两种更简单的操作：过滤和计数，看似有化简为繁之嫌——例 3-1 中只含一个 `for` 循环，两种操作是否意味着需要两次循环？事实上，类库设计精妙，只需对艺术家列表迭代一次。

通常，在 Java 中调用一个方法，计算机会随即执行操作：比如，`System.out.println("Hello World");` 会在终端上输出一条信息。`Stream` 里的一些方法却略有不同，它们虽是普通的 Java 方法，但返回的 `Stream` 对象却不是一个新集合，而是创建新集合的配方。现在，尝试思考一下例 3-4 中代码的作用，一时毫无头绪也没关系，稍后会详细解释。

例 3-4　只过滤，不计数

```
allArtists.stream()
          .filter(artist -> artist.isFrom("London"));
```

这行代码并未做什么实际性的工作，filter 只刻画出了 Stream，但没有产生新的集合。像 filter 这样只描述 Stream，最终不产生新集合的方法叫作*惰性求值方法*；而像 count 这样最终会从 Stream 产生值的方法叫作*及早求值方法*。

如果在过滤器中加入一条 println 语句，来输出艺术家的名字，就能轻而易举地看出其中的不同。例 3-5 对例 3-4 作了一些修改，加入了输出语句。运行这段代码，程序不会输出任何信息！

例 3-5　由于使用了惰性求值，没有输出艺术家的名字

```
allArtists.stream()
          .filter(artist -> {
              System.out.println(artist.getName());
              return artist.isFrom("London");
           });
```

如果将同样的输出语句加入一个拥有终止操作的流，如例 3-3 中的计数操作，艺术家的名字就会被输出（见例 3-6）。

例 3-6　输出艺术家的名字

```
long count = allArtists.stream()
                       .filter(artist -> {
                           System.out.println(artist.getName());
                           return artist.isFrom("London");
                       })
                       .count();
```

以披头士乐队的成员作为艺术家列表，运行上述程序，命令行里输出的内容如例 3-7 所示。

例 3-7　显示披头士乐队成员名单的示例输出

```
John Lennon
Paul McCartney
George Harrison
Ringo Starr
```

判断一个操作是惰性求值还是及早求值很简单：只需看它的返回值。如果返回值是 Stream，那么是惰性求值；如果返回值是另一个值或为空，那么就是及早求值。使用这些操作的理想方式就是形成一个惰性求值的链，最后用一个及早求值的操作返回想要的结果，这正是它的合理之处。计数的示例也是这样运行的，但这只是最简单的情况：只含两步操作。

整个过程和*建造者*模式有共通之处。建造者模式使用一系列操作设置属性和配置，最后调用一个 build 方法，这时，对象才被真正创建。

读者一定会问："为什么要区分惰性求值和及早求值？"只有在对需要什么样的结果和操

作有了更多了解之后，才能更有效率地进行计算。例如，如果要找出大于 10 的第一个数字，那么并不需要和所有元素去做比较，只要找出第一个匹配的元素就够了。这也意味着可以在集合类上级联多种操作，但迭代只需一次。

3.3 常用的流操作

为了更好地理理解 Stream API，掌握一些常用的 Stream 操作十分必要。除此处讲述的几种重要操作之外，该 API 的 Javadoc 中还有更多信息。

3.3.1 collect(toList())

> collect(toList()) 方法由 Stream 里的值生成一个列表，是一个及早求值操作。

Stream 的 of 方法使用一组初始值生成新的 Stream。事实上，collect 的用法不仅限于此，它是一个非常通用的强大结构，第 5 章将详细介绍它的其他用途。下面是使用 collect 方法的一个例子：

```
List<String> collected = Stream.of("a", "b", "c")  ❶
                               .collect(Collectors.toList());  ❷

assertEquals(Arrays.asList("a", "b", "c"), collected);  ❸
```

这段程序展示了如何使用 collect(toList()) 方法从 Stream 中生成一个列表。如上文所述，由于很多 Stream 操作都是惰性求值，因此调用 Stream 上一系列方法之后，还需要最后再调用一个类似 collect 的及早求值方法。

这个例子也展示了本节中所有示例代码的通用格式。首先由列表生成一个 Stream ❶，然后进行一些 Stream 上的操作，继而是 collect 操作，由 Stream 生成列表❷，最后使用断言判断结果是否和预期一致❸。

形象一点儿的话，可以将 Stream 想象成汉堡，将最前和最后对 Stream 操作的方法想象成两片面包，这两片面包帮助我们认清操作的起点和终点。

3.3.2 map

> 如果有一个函数可以将一种类型的值转换成另外一种类型，map 操作就可以使用该函数，将一个流中的值转换成一个新的流。

读者可能已经注意到，以前编程时或多或少使用过类似 map 的操作。比如编写一段 Java 代码，将一组字符串转换成对应的大写形式。在一个循环中，对每个字符串调用 toUppercase 方法，然后将得到的结果加入一个新的列表。代码如例 3-8 所示。

例 3-8　使用 for 循环将字符串转换为大写

```
List<String> collected = new ArrayList<>();
for (String string : asList("a", "b", "hello")) {
    String uppercaseString = string.toUpperCase();
    collected.add(uppercaseString);
}

assertEquals(asList("A", "B", "HELLO"), collected);
```

如果你经常实现例 3-8 中这样的 for 循环，就不难猜出 map 是 Stream 上最常用的操作之一（如图 3-3 所示）。例 3-9 展示了如何使用新的流框架将一组字符串转换成大写形式。

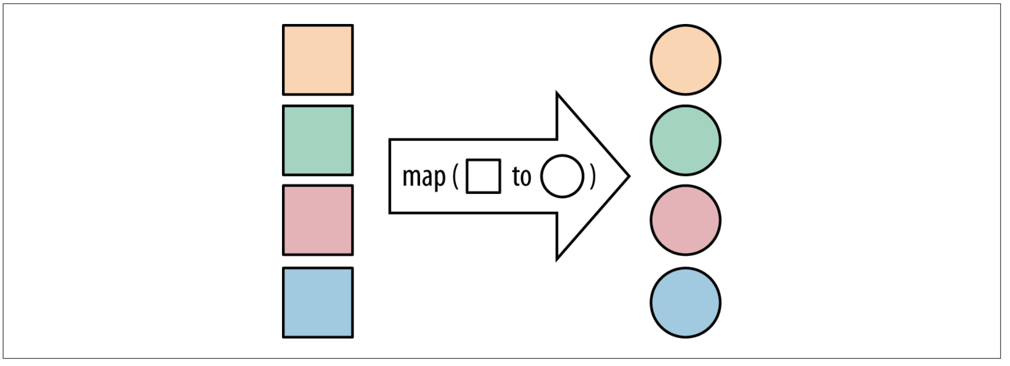


图 3-3：map 操作

例 3-9　使用 map 操作将字符串转换为大写形式

```
List<String> collected = Stream.of("a", "b", "hello")
                               .map(string -> string.toUpperCase())  ❶
                               .collect(toList());

assertEquals(asList("A", "B", "HELLO"), collected);
```

传给 map ❶ 的 Lambda 表达式只接受一个 String 类型的参数，返回一个新的 String。参数和返回值不必属于同一种类型，但是 Lambda 表达式必须是 Function 接口的一个实例（如图 3-4 所示），Function 接口是只包含一个参数的普通函数接口。

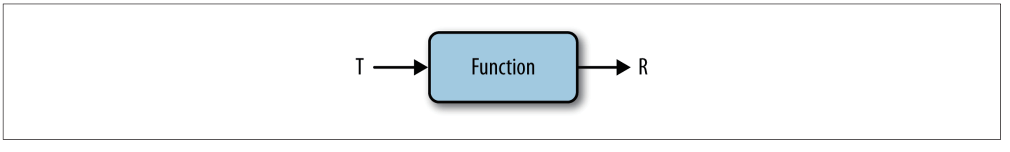


图 3-4：Function 接口

3.3.3 filter

遍历数据并检查其中的元素时，可尝试使用 Stream 中提供的新方法 filter（如图 3-5 所示）。

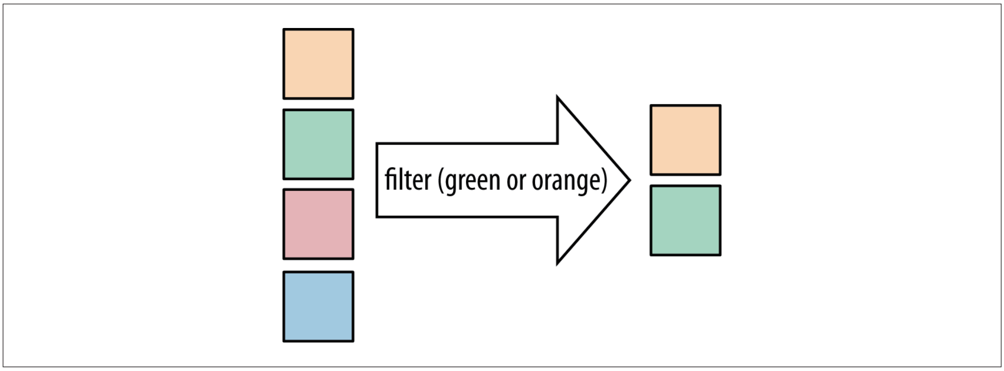


图 3-5：filter 操作

上面就是一个使用 filter 的例子，如果你已熟悉这一概念，也可以选择跳过本节。啊哈！您还没跳过本节？那太好了，我们一起来看看这个方法有什么用。假设要找出一组字符串中以数字开头的字符串，比如字符串 "1abc" 和 "abc"，其中 "1abc" 就是符合条件的字符串。可以使用一个 for 循环，内部用 if 条件语句判断字符串的第一个字符来解决这个问题，代码如例 3-10 所示。

例 3-10　使用循环遍历列表，使用条件语句做判断

```
List<String> beginningWithNumbers = new ArrayList<>();
for(String value : asList("a", "1abc", "abc1")) {
    if (isDigit(value.charAt(0))) {
        beginningWithNumbers.add(value);
    }
}

assertEquals(asList("1abc"), beginningWithNumbers);
```

你可能已经写过很多类似的代码：这被称为 filter 模式。该模式的核心思想是保留 Stream 中的一些元素，而过滤掉其他的。例 3-11 展示了如何使用函数式风格编写相同的代码。

例 3-11　函数式风格

```
List<String> beginningWithNumbers
  = Stream.of("a", "1abc", "abc1")
        .filter(value -> isDigit(value.charAt(0)))
        .collect(toList());
```

```java
    assertEquals(asList("1abc"), beginningWithNumbers);
```

和 `map` 很像，`filter` 接受一个函数作为参数，该函数用 Lambda 表达式表示。该函数和前面示例中 `if` 条件判断语句的功能一样，如果字符串首字母为数字，则返回 `true`。若要重构遗留代码，`for` 循环中的 `if` 条件语句就是一个很强的信号，可用 `filter` 方法替代。

由于此方法和 `if` 条件语句的功能相同，因此其返回值肯定是 `true` 或者 `false`。经过过滤，`Stream` 中符合条件的，即 Lambda 表达式值为 `true` 的元素被保留下来。该 Lambda 表达式的函数接口正是前面章节中介绍过的 `Predicate`（如图 3-6 所示）。

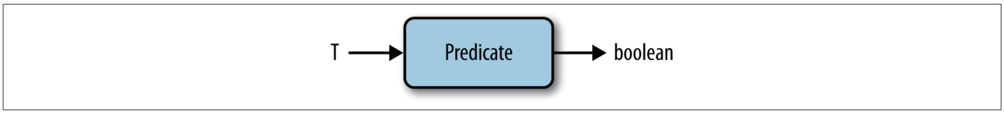


图 3-6：`Predicate` 接口

3.3.4　flatMap

> `flatMap` 方法可用 `Stream` 替换值，然后将多个 Stream 连接成一个 `Stream`（如图 3-7 所示）。

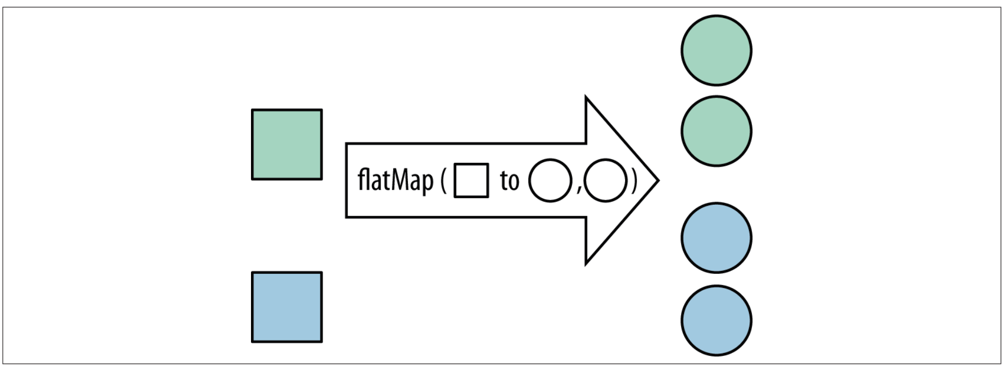


图 3-7：`flatMap` 操作

前面已介绍过 `map` 操作，它可用一个新的值代替 `Stream` 中的值。但有时，用户希望让 `map` 操作有点变化，生成一个新的 `Stream` 对象取而代之。用户通常不希望结果是一连串的流，此时 `flatMap` 最能派上用场。

我们看一个简单的例子。假设有一个包含多个列表的流，现在希望得到所有数字的序列。该问题的一个解法如例 3-12 所示。

例 3-12　包含多个列表的 Stream

```
List<Integer> together = Stream.of(asList(1, 2), asList(3, 4))
                               .flatMap(numbers -> numbers.stream())
                               .collect(toList());

assertEquals(asList(1, 2, 3, 4), together);
```

调用 stream 方法，将每个列表转换成 Stream 对象，其余部分由 flatMap 方法处理。flatMap 方法的相关函数接口和 map 方法的一样，都是 Function 接口，只是方法的返回值限定为 Stream 类型罢了。

3.3.5　max和min

Stream 上常用的操作之一是求最大值和最小值。Stream API 中的 max 和 min 操作足以解决这一问题。例 3-13 是查找专辑中最短曲目所用的代码，展示了如何使用 max 和 min 操作。为了方便检查程序结果是否正确，代码片段中罗列了专辑中的曲目信息，我承认，这张专辑是有点冷门。

例 3-13　使用 Stream 查找最短曲目

```
List<Track> tracks = asList(new Track("Bakai", 524),
                            new Track("Violets for Your Furs", 378),
                            new Track("Time Was", 451));

Track shortestTrack = tracks.stream()
                            .min(Comparator.comparing(track -> track.getLength()))
                            .get();

assertEquals(tracks.get(1), shortestTrack);
```

查找 Stream 中的最大或最小元素，首先要考虑的是用什么作为排序的指标。以查找专辑中的最短曲目为例，排序的指标就是曲目的长度。

为了让 Stream 对象按照曲目长度进行排序，需要传给它一个 Comparator 对象。Java 8 提供了一个新的静态方法 comparing，使用它可以方便地实现一个比较器。放在以前，我们需要比较两个对象的某项属性的值，现在只需要提供一个存取方法就够了。本例中使用 getLength 方法。

花点时间研究一下 comparing 方法是值得的。实际上这个方法接受一个函数并返回另一个函数。我知道，这听起来像句废话，但是却很有用。这个方法本该早已加入 Java 标准库，但由于匿名内部类可读性差且书写冗长，一直未能实现。现在有了 Lambda 表达式，代码变得简洁易懂。

此外，还可以调用空 Stream 的 max 方法，返回 Optional 对象。Optional 对象有点陌生，它代表一个可能存在也可能不存在的值。如果 Stream 为空，那么该值不存在，如果不为空，则该值存在。先不必细究，4.10 节将详细讲述 Optional 对象，现在唯一需要记住的是，通过调用 get 方法可以取出 Optional 对象中的值。

3.3.6 通用模式

max 和 min 方法都属于更通用的一种编程模式。要看到这种编程模式，最简单的方法是使用 for 循环重写例 3-13 中的代码。例 3-14 和例 3-13 的功能一样，都是查找专辑中的最短曲目，但是使用了 for 循环。

例 3-14 使用 for 循环查找最短曲目

```
List<Track> tracks = asList(new Track("Bakai", 524),
                            new Track("Violets for Your Furs", 378),
                            new Track("Time Was", 451));

Track shortestTrack = tracks.get(0);
for (Track track : tracks) {
    if (track.getLength() < shortestTrack.getLength()) {
        shortestTrack = track;
    }
}

assertEquals(tracks.get(1), shortestTrack);
```

这段代码先使用列表中的第一个元素初始化变量 shortestTrack，然后遍历曲目列表，如果找到更短的曲目，则更新 shortestTrack，最后变量 shortestTrack 保存的正是最短曲目。程序员们无疑已写过成千上万次这样的 for 循环，其中很多都属于这个模式。例 3-15 中的伪代码体现了通用模式的特点。

例 3-15 reduce 模式

```
Object accumulator = initialValue;
for(Object element : collection) {
    accumulator = combine(accumulator, element);
}
```

首先赋给 accumulator 一个初始值：initialValue，然后在循环体中，通过调用 combine 函数，拿 accumulator 和集合中的每一个元素做运算，再将运算结果赋给 accumulator，最后 accumulator 的值就是想要的结果。

这个模式中的两个可变项是 initialValue 初始值和 combine 函数。在例 3-14 中，我们选列表中的第一个元素为初始值，但也不必需如此。为了找出最短曲目，combine 函数返回当前元素和 accumulator 中较短的那个。

接下来看一下 Stream API 中的 reduce 操作是怎么工作的。

3.3.7 reduce

reduce 操作可以实现从一组值中生成一个值。在上述例子中用到的 count、min 和 max 方法，因为常用而被纳入标准库中。事实上，这些方法都是 reduce 操作。

图 3-8 展示了如何通过 reduce 操作对 Stream 中的数字求和。以 0 作起点——一个空

Stream 的求和结果，每一步都将 Stream 中的元素累加至 accumulator，遍历至 Stream 中的最后一个元素时，accumulator 的值就是所有元素的和。

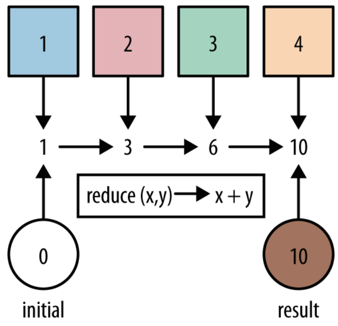


图 3-8：使用 reduce 操作实现累加

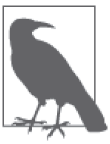
例 3-16 中的代码展示了这一过程。Lambda 表达式就是 reducer，它执行求和操作，有两个参数：传入 Stream 中的当前元素和 acc。将两个参数相加，acc 是累加器，保存着当前的累加结果。

例 3-16　使用 reduce 求和

```
int count = Stream.of(1, 2, 3)
                  .reduce(0, (acc, element) -> acc + element);

assertEquals(6, count);
```

Lambda 表达式的返回值是最新的 acc，是上一轮 acc 的值和当前元素相加的结果。reducer 的类型是第 2 章已介绍过的 BinaryOperator。



4.2 节将介绍另外一种标准类库内置的求和方法，在实际生产环境中，应该使用那种方式，而不是使用像上面这个例子中的代码。

表 3-1 显示了求和过程中的中间值。事实上，可以将 reduce 操作展开，得到例 3-17 这样形式的代码。

例 3-17　展开 reduce 操作

```
BinaryOperator<Integer> accumulator = (acc, element) -> acc + element;
int count = accumulator.apply(
                accumulator.apply(
                    accumulator.apply(0, 1),
                2),
            3);
```

表3-1　reduce过程的中间值

元　素	acc	结　果
N/A	N/A	0
1	0	1
2	1	3
3	3	6

例 3-18 是可实现同样功能的命令式 Java 代码，从中可清楚看出函数式编程和命令式编程的区别。

例 3-18　使用命令式编程方式求和

```
int acc = 0;
for (Integer element : asList(1, 2, 3)) {
    acc = acc + element;
}
assertEquals(6, acc);
```

在命令式编程方式下，每一次循环将集合中的元素和累加器相加，用相加后的结果更新累加器的值。对于集合来说，循环在外部，且需要手动更新变量。

3.3.8　整合操作

`Stream` 接口的方法如此之多，有时会让人难以选择，像闯入一个迷宫，不知道该用哪个方法更好。本节将举例说明如何将问题分解为简单的 `Stream` 操作。

第一个要解决的问题是，找出某张专辑上所有乐队的国籍。艺术家列表里既有个人，也有乐队。利用一点领域知识，假定一般乐队名以定冠词 The 开头。当然这不是绝对的，但也差不多。

需要注意的是，这个问题绝不是简单地调用几个 API 就足以解决。这既不是使用 `map` 将一组值映射为另一组值，也不是过滤，更不是将 `Stream` 中的元素最终归约为一个值。首先，可将这个问题分解为如下几个步骤。

1. 找出专辑上的所有表演者。
2. 分辨出哪些表演者是乐队。
3. 找出每个乐队的国籍。
4. 将找出的国籍放入一个集合。

现在，找出每一步对应的 `Stream` API 就相对容易了：

1. `Album` 类有个 `getMusicians` 方法，该方法返回一个 `Stream` 对象，包含整张专辑中所有的表演者；

2. 使用 filter 方法对表演者进行过滤，只保留乐队；
3. 使用 map 方法将乐队映射为其所属国家；
4. 使用 collect(Collectors.toList()) 方法将国籍放入一个列表。

最后，整合所有的操作，就得到如下代码：

```
Set<String> origins = album.getMusicians()
                           .filter(artist -> artist.getName().startsWith("The"))
                           .map(artist -> artist.getNationality())
                           .collect(toSet());
```

这个例子将 Stream 的链式操作展现得淋漓尽致，调用 getMusicians、filter 和 map 方法都返回 Stream 对象，因此都属于惰性求值，而 collect 方法属于及早求值。map 方法接受一个 Lambda 表达式，使用该 Lambda 表达式对 Stream 上的每个元素做映射，形成一个新的 Stream。

这个问题处理起来很方便，使用 getMusicians 方法获取专辑上的艺术家列表时得到的是一个 Stream 对象。然而，处理其他实际遇到的问题时未必也能如此方便，很可能没有方法可以返回一个 Stream 对象，反而得到像 List 或 Set 这样的集合类。别担心，只要调用 List 或 Set 的 stream 方法就能得到一个 Stream 对象。

现在或许是个思考的好机会，你真的需要对外暴露一个 List 或 Set 对象吗？可能一个 Stream 工厂才是更好的选择。通过 Stream 暴露集合的最大优点在于，它很好地封装了内部实现的数据结构。仅暴露一个 Stream 接口，用户在实际操作中无论如何使用，都不会影响内部的 List 或 Set。

同时这也鼓励用户在编程中使用更现代的 Java 8 风格。不必一蹴而就，可以对已有代码渐进性地重构，保留原有的取值函数，添加返回 Stream 对象的函数，时间长了，就可以删掉所有返回 List 或 Set 的取值函数。清理了所有遗留代码之后，这种重构方式让人感觉棒极了！

3.4　重构遗留代码

为了进一步阐释如何重构遗留代码，本节将举例说明如何将一段使用循环进行集合操作的代码，重构成基于 Stream 的操作。重构过程中的每一步都能确保代码通过单元测试，当然你也可以自行实际操作一遍，体验并验证。

假定选定一组专辑，找出其中所有长度大于 1 分钟的曲目名称。例 3-19 是遗留代码，首先初始化一个 Set 对象，用来保存找到的曲目名称。然后使用 for 循环遍历所有专辑，每次循环中再使用一个 for 循环遍历每张专辑上的每首曲目，检查其长度是否大于 60 秒，如果是，则将该曲目名称加入 Set 对象。

例 3-19　遗留代码：找出长度大于 1 分钟的曲目

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    for(Album album : albums) {
        for (Track track : album.getTrackList()) {
            if (track.getLength() > 60) {
                String name = track.getName();
                trackNames.add(name);
            }
        }
    }
    return trackNames;
}
```

如果仔细阅读上面的这段代码，就会发现几组嵌套的循环。仅通过阅读这段代码很难看出它的编写目的，那就来重构一下（使用流来重构该段代码的方式很多，下面介绍的只是其中一种。事实上，对 Stream API 越熟悉，就越不需要细分步骤。之所以在示例中一步一步地重构，完全是出于帮助大家学习的目的，在工作中无需这样做）。

第一步要修改的是 for 循环。首先使用 Stream 的 forEach 方法替换掉 for 循环，但还是暂时保留原来循环体中的代码，这是在重构时非常方便的一个技巧。调用 stream 方法从专辑列表中生成第一个 Stream，同时不要忘了在上一节已介绍过，getTracks 方法本身就返回一个 Stream 对象。经过第一步重构后，代码如例 3-20 所示。

例 3-20　重构的第一步：找出长度大于 1 分钟的曲目

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
          .forEach(album -> {
              album.getTracks()
                   .forEach(track -> {
                       if (track.getLength() > 60) {
                           String name = track.getName();
                           trackNames.add(name);
                       }
                   });
          });
    return trackNames;
}
```

在重构的第一步中，虽然使用了流，但是并没有充分发挥它的作用。事实上，重构后的代码还不如原来的代码好——天哪！因此，是时候引入一些更符合流风格的代码了，最内层的 forEach 方法正是主要突破口。

最内层的 forEach 方法有三个功用：找出长度大于 1 分钟的曲目，得到符合条件的曲目名称，将曲目名称加入集合 Set。这就意味着需要三项 Stream 操作：找出满足某种条件的曲目是 filter 的功能，得到曲目名称则可用 map 达成，终结操作可使用 forEach 方法将曲目

名称加入一个集合。用以上三项 Stream 操作将内部的 forEach 方法拆分后，代码如例 3-21 所示。

例 3-21　重构的第二步：找出长度大于 1 分钟的曲目

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
          .forEach(album -> {
              album.getTracks()
                   .filter(track -> track.getLength() > 60)
                   .map(track -> track.getName())
                   .forEach(name -> trackNames.add(name));
          });
    return trackNames;
}
```

现在用更符合流风格的操作替换了内层的循环，但代码看起来还是冗长繁琐。将各种流嵌套起来并不理想，最好还是用干净整洁的顺序调用一些方法。

理想的操作莫过于找到一种方法，将专辑转化成一个曲目的 Stream。众所周知，任何时候想转化或替代代码，都该使用 map 操作。这里将使用比 map 更复杂的 flatMap 操作，把多个 Stream 合并成一个 Stream 并返回。将 forEach 方法替换成 flatMap 后，代码如例 3-22 所示。

例 3-22　重构的第三步：找出长度大于 1 分钟的曲目

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();

    albums.stream()
          .flatMap(album -> album.getTracks())
          .filter(track -> track.getLength() > 60)
          .map(track -> track.getName())
          .forEach(name -> trackNames.add(name));

    return trackNames;
}
```

上面的代码中使用一组简洁的方法调用替换掉两个嵌套的 for 循环，看起来清晰很多。然而至此并未结束，仍需手动创建一个 Set 对象并将元素加入其中，但我们希望看到的是整个计算任务由一连串的 Stream 操作完成。

到目前为止，虽然还未展示转换的方法，但已有类似的操作。就像使用 collect(Collectors.toList()) 可以将 Stream 中的值转换成一个列表，使用 collect(Collectors.toSet()) 可以将 Stream 中的值转换成一个集合。因此，将最后的 forEach 方法替换为 collect，并删掉变量 trackNames，代码如例 3-23 所示。

例 3-23　重构的第四步：找出长度大于 1 分钟的曲目

```
public Set<String> findLongTracks(List<Album> albums) {
```

```
        return albums.stream()
                     .flatMap(album -> album.getTracks())
                     .filter(track -> track.getLength() > 60)
                     .map(track -> track.getName())
                     .collect(toSet());
    }
```

简而言之，选取一段遗留代码进行重构，转换成使用流风格的代码。最初只是简单地使用流，但没有引入任何有用的流操作。随后通过一系列重构，最终使代码更符合使用流的风格。在上述步骤中我们没有提到一个重点，即编写示例代码的每一步都要进行单元测试，保证代码能够正常工作。重构遗留代码时，这样做很有帮助。

3.5　多次调用流操作

用户也可以选择每一步强制对函数求值，而不是将所有的方法调用链接在一起，但是，最好不要如此操作。例 3-24 展示了如何用如上述不建议的编码风格来找出专辑上所有演出乐队的国籍，例 3-25 则是之前的代码，放在一起方便比较。

例 3-24　误用 Stream 的例子

```
List<Artist> musicians = album.getMusicians()
                              .collect(toList());

List<Artist> bands = musicians.stream()
                              .filter(artist -> artist.getName().startsWith("The"))
                              .collect(toList());

Set<String> origins = bands.stream()
                           .map(artist -> artist.getNationality())
                           .collect(toSet());
```

例 3-25　符合 Stream 使用习惯的链式调用

```
Set<String> origins = album.getMusicians()
                           .filter(artist -> artist.getName().startsWith("The"))
                           .map(artist -> artist.getNationality())
                           .collect(toSet());
```

例 3-24 所示代码和流的链式调用相比有如下缺点：

- 代码可读性差，样板代码太多，隐藏了真正的业务逻辑；
- 效率差，每一步都要对流及早求值，生成新的集合；
- 代码充斥一堆垃圾变量，它们只用来保存中间结果，除此之外毫无用处；
- 难于自动并行化处理。

当然，刚开始写基于流的程序时，这样的情况在所难免。但是如果发现自己经常写出这样的代码，就要反思能否将代码重构得更加简洁易读。

> 如果此时还不习惯 `Stream` API 中大量的链式操作，也很正常。随着练习时间增加，经验也会越来越丰富，这些概念理解起来也更加自然。因此，尚未习惯不能成为拆开链式操作、写出形如例 3-24 中代码的理由。像使用建造者模式那样，按规则写出每一行代码，可以帮助用户慢慢习惯这种链式操作。

3.6 高阶函数

本章中不断出现被函数式编程程序员称为*高阶函数*的操作。高阶函数是指接受另外一个函数作为参数，或返回一个函数的函数。高阶函数不难辨认：看函数签名就够了。如果函数的参数列表里包含函数接口，或该函数返回一个函数接口，那么该函数就是高阶函数。

`map` 是一个高阶函数，因为它的 `mapper` 参数是一个函数。事实上，本章介绍的 `Stream` 接口中几乎所有的函数都是高阶函数。之前的排序例子中还用到了 `comparing` 函数，它接受一个函数作为参数，获取相应的值，同时返回一个 `Comparator`。`Comparator` 可能会被误认为是一个对象，但它有且只有一个抽象方法，所以实际上是一个函数接口。

事实上，可以大胆断言，`Comparator` 实际上应该是个函数，但是那时的 Java 只有对象，因此才造出了一个类，一个匿名类。成为对象实属巧合，函数接口向正确的方向迈出了一步。

3.7 正确使用Lambda表达式

刚开始介绍 Lambda 表达式时，以能够输出一些信息的回调函数为示例。回调函数是一个合法的 Lambda 表达式，但并不能真正帮助用户写出更简单、更抽象的代码，因为它仍然在指挥计算机执行一个操作。清理掉样板代码很有帮助，但 Java 8 引入的 Lambda 表达式的作用远不止这些。

本章介绍的概念能够帮助用户写出更简单的代码，因为这些概念描述了数据上的操作，明确了要达成*什么转化*，而不是说明*如何转化*。这种方式写出的代码，潜在的缺陷更少，更直接地表达了程序员的意图。

明确要达成*什么转化*，而不是说明*如何转化*的另外一层含义在于写出的函数没有副作用。这一点非常重要，这样只通过函数的返回值就能充分理解函数的全部作用。

没有副作用的函数不会改变程序或外界的状态。本书中的第一个 Lambda 表达式示例是有副作用的，它向控制台输出了信息——一个*可观测到*的副作用。下面的代码有没有副作用？

```java
private ActionEvent lastEvent;

private void registerHandler() {
    button.addActionListener((ActionEvent event) -> {
        this.lastEvent = event;
```

```
        });
    }
```

这里将参数 `event` 保存至成员变量 `lastEvent`。给变量赋值也是一种副作用，而且更难察觉。在程序的输出中可能很难直接观察到，但是它的确更改了程序的状态。Java 在这方面有局限性，例如下面这段代码，赋值给一个局部变量 `localEvent`：

```
ActionEvent localEvent = null;
button.addActionListener(event -> {
    localEvent = event;
});
```

这段代码试图将 `event` 赋给一个局部变量，它无法通过编译，但绝非编写错误。这实际上是语言的设计者有意为之，用以鼓励用户使用 Lambda 表达式获取值而不是变量。获取值使用户更容易写出没有副作用的代码。如第二章所述，在 Lambda 表达式中使用局部变量，可以不使用 `final` 关键字，但局部变量在既成事实上必须是 `final` 的。

无论何时，将 Lambda 表达式传给 `Stream` 上的高阶函数，都应该尽量避免副作用。唯一的例外是 `forEach` 方法，它是一个终结方法。

3.8 要点回顾

- 内部迭代将更多控制权交给了集合类。
- 和 `Iterator` 类似，`Stream` 是一种内部迭代方式。
- 将 Lambda 表达式和 `Stream` 上的方法结合起来，可以完成很多常见的集合操作。

3.9 练习

练习的答案可以在 GitHub 代码仓库（https://github.com/RichardWarburton/java-8-Lambdas-exercises）中找到。

1. 常用流操作。实现如下函数：

   a. 编写一个求和函数，计算流中所有数之和。例如，`int addUp(Stream<Integer> numbers)`；

   b. 编写一个函数，接受艺术家列表作为参数，返回一个字符串列表，其中包含艺术家的姓名和国籍；

   c. 编写一个函数，接受专辑列表作为参数，返回一个由最多包含 3 首歌曲的专辑组成的列表。

2. 迭代。修改如下代码，将外部迭代转换成内部迭代：

```
int totalMembers = 0;
for (Artist artist : artists) {
    Stream<Artist> members = artist.getMembers();
    totalMembers += members.count();
}
```

3. 求值。根据 `Stream` 方法的签名，判断其是惰性求值还是及早求值。

   a. `boolean anyMatch(Predicate<? super T> predicate);`

   b. `Stream<T> limit(long maxSize);`

4. 高阶函数。下面的 `Stream` 函数是高阶函数吗？为什么？

   a. `boolean anyMatch(Predicate<? super T> predicate);`

   b. `Stream<T> limit(long maxSize);`

5. 纯函数。下面的 Lambda 表达式有无副作用，或者说它们是否更改了程序状态？

```
x -> x + 1
```

示例代码如下所示：

```
AtomicInteger count = new AtomicInteger(0);
List<String> origins = album.musicians()
                             .forEach(musician -> count.incAndGet();)
```

   a. 上述示例代码中传入 `forEach` 方法的 Lambda 表达式。

6. 计算一个字符串中小写字母的个数（提示：参阅 `String` 对象的 `chars` 方法）。

7. 在一个字符串列表中，找出包含最多小写字母的字符串。对于空列表，返回 `Optional<String>` 对象。

3.10　进阶练习

1. 只用 `reduce` 和 Lambda 表达式写出实现 `Stream` 上的 `map` 操作的代码，如果不想返回 `Stream`，可以返回一个 `List`。

2. 只用 `reduce` 和 Lambda 表达式写出实现 `Stream` 上的 `filter` 操作的代码，如果不想返回 `Stream`，可以返回一个 `List`。

第 4 章
类库

前 3 章讨论了如何编写 Lambda 表达式，接下来将详细阐述另一个重要方面：如何使用 Lambda 表达式。即使不需要编写像 `Stream` 这样重度使用函数式编程风格的类库，学会如何使用 Lambda 表达式也是非常重要的。即使一个最简单的应用，也可能会因为代码即数据的函数式编程风格而受益。

Java 8 中的另一个变化是引入了*默认方法*和接口的静态方法，它改变了人们认识类库的方式，接口中的方法也可以包含代码体了。

本章还对前 3 章疏漏的知识点进行补充，比如，Lambda 表达式方法重载的工作原理、基本类型的使用方法等。使用 Lambda 表达式编写程序时，掌握这些知识非常重要。

4.1　在代码中使用Lambda表达式

2.5 节介绍了如何赋予 Lambda 表达式函数接口的类型，以及该类型的推导方式。从调用 Lambda 表达式的代码的角度来看，它和调用一个普通接口方法没什么区别。

让我们来看一个日志系统中的具体案例。在 `slf4j` 和 `log4j` 等几种常用的日志系统中，有一些记录日志的方法，当日志级别不低于某个固定级别时就会开始记录日志。如此一来，在日志框架中设置类似 `void debug(String message)` 这样的方法，当级别为 `debug` 时，它们就开始记录日志消息。

问题在于，频繁计算消息是否应该记录日志会对系统性能产生影响。程序员通过显式调用 `isDebugEnabled` 方法来优化系统性能，如例 4-1 所示。即使直接调用 `debug` 方法能省去记

录文本信息，也仍然需要调用 expensiveOperation 方法，并且需要将执行结果和已有字符串连接起来，因此，使用 if 语句显式判断，可以让程序跑得更快。

例 4-1　使用 isDebugEnabled 方法降低日志性能开销

```
Logger logger = new Logger();
if (logger.isDebugEnabled()) {
    logger.debug("Look at this: " + expensiveOperation());
}
```

这里我们想做的是传入一个 Lambda 表达式，生成一条用作日志信息的字符串。只有日志级别在调试或以上级别时，才会执行该 Lambda 表达式。使用这个方式重写上面的代码，如例 4-2 所示：

例 4-2　使用 Lambda 表达式简化日志代码

```
Logger logger = new Logger();
logger.debug(() -> "Look at this: " + expensiveOperation());
```

那么在 Logger 类中该方法是如何实现的呢？从类库的角度看，我们可以使用内置的 Supplier 函数接口，它只有一个 get 方法。然后通过调用 isDebugEnabled 判断是否需要记录日志，是否需要调用 get 方法，如果需要，就调用 get 方法并将结果传给 debug 方法。由此产生的代码如例 4-3 所示。

例 4-3　启用 Lambda 表达式实现的日志记录器

```
public void debug(Supplier<String> message) {
    if (isDebugEnabled()) {
        debug(message.get());
    }
}
```

调用 get() 方法，相当于调用传入的 Lambda 表达式。这种方式也能和匿名内部类一起工作，如果用户暂时无法升级到 Java 8，这种方式可以实现向后兼容。

值得注意的是，不同的函数接口有不同的方法。如果使用 Predicate，就应该调用 test 方法，如果使用 Function，就应该调用 apply 方法。

4.2　基本类型

以上部分还没有用到*基本类型*。在 Java 中，有一些相伴的类型，比如 int 和 Integer——前者是基本类型，后者是*装箱类型*。基本类型内建在语言和运行环境中，是基本的程序构建模块；而装箱类型属于普通的 Java 类，只不过是对基本类型的一种封装。

Java 的泛型是基于对泛型参数类型的*擦除*——换句话说，假设它是 Object 对象的实例——因此只有装箱类型才能作为泛型参数。这就解释了为什么在 Java 中想要一个包含整型值的列表 List<int>，实际上得到的却是一个包含整型对象的列表 List<Integer>。

麻烦的是，由于装箱类型是对象，因此在内存中存在额外开销。比如，整型在内存中占用 4 字节，整型对象却要占用 16 字节。这一情况在数组上更加严重，整型数组中的每个元素只占用基本类型的内存，而整型对象数组中，每个元素都是内存中的一个指针，指向 Java 堆中的某个对象。在最坏的情况下，同样大小的数组，`Integer[]` 要比 `int[]` 多占用 6 倍内存。

将基本类型转换为装箱类型，称为装箱，反之则称为拆箱，两者都需要额外的计算开销。对于需要大量数值运算的算法来说，装箱和拆箱的计算开销，以及装箱类型占用的额外内存，会明显减缓程序的运行速度。

为了减小这些性能开销，`Stream` 类的某些方法对基本类型和装箱类型做了区分。图 4-1 所示的高阶函数 `mapToLong` 和其他类似函数即为该方面的一个尝试。在 Java 8 中，仅对整型、长整型和双浮点型做了特殊处理，因为它们在数值计算中用得最多，特殊处理后的系统性能提升效果最明显。



图 4-1：`ToLongFunction`

对基本类型做特殊处理的方法在命名上有明确的规范。如果方法返回类型为基本类型，则在基本类型前加 `To`，如图 4-1 中的 `ToLongFunction`。如果参数是基本类型，则不加前缀只需类型名即可，如图 4-2 中的 `LongFunction`。如果高阶函数使用基本类型，则在操作后加后缀 `To` 再加基本类型，如 `mapToLong`。



图 4-2：`LongFunction`

这些基本类型都有与之对应的 Stream，以基本类型名为前缀，如 `LongStream`。事实上，`mapToLong` 方法返回的不是一个一般的 `Stream`，而是一个特殊处理的 `Stream`。在这个特殊的 `Stream` 中，`map` 方法的实现方式也不同，它接受一个 `LongUnaryOperator` 函数，将一个长整型值映射成另一个长整型值，如图 4-3 所示。通过一些高阶函数装箱方法，如 `mapToObj`，也可以从一个基本类型的 `Stream` 得到一个装箱后的 `Stream`，如 `Stream<Long>`。



图 4-3：`LongUnaryOperator`

如有可能，应尽可能多地使用对基本类型做过特殊处理的方法，进而改善性能。这些特殊的 Stream 还提供额外的方法，避免重复实现一些通用的方法，让代码更能体现出数值计算的意图。例 4-4 展示了如何使用这些方法：

例 4-4　使用 summaryStatistics 方法统计曲目长度

```
public static void printTrackLengthStatistics(Album album) {
    IntSummaryStatistics trackLengthStats
            = album.getTracks()
                   .mapToInt(track -> track.getLength())
                   .summaryStatistics();
    System.out.printf("Max: %d, Min: %d, Ave: %f, Sum: %d",
                      trackLengthStats.getMax(),
                      trackLengthStats.getMin(),
                      trackLengthStats.getAverage(),
                      trackLengthStats.getSum());
}
```

例 4-4 向控制台输出曲目长度的一系列统计信息。无需手动计算这些信息，这里使用对基本类型进行特殊处理的方法 mapToInt，将每首曲目映射为曲目长度。因为该方法返回一个 IntStream 对象，它包含一个 summaryStatistics 方法，这个方法能计算出各种各样的统计值，如 IntStream 对象内所有元素中的最小值、最大值、平均值以及数值总和。

这些统计值在所有特殊处理的 Stream，如 DoubleStream、LongStream 中都可以得出。如无需全部的统计值，也可分别调用 min、max、average 或 sum 方法获得单个的统计值，同样，三种基本类型对应的特殊 Stream 也都包含这些方法。

4.3　重载解析

在 Java 中可以重载方法，造成多个方法有相同的方法名，但签名确不一样。这在推断参数类型时会带来问题，因为系统可能会推断出多种类型。这时，javac 会挑出最具体的类型。如例 4-5 中的方法调用在选择例 4-6 中定义的重载方法时，输出 String，而不是 Object。

例 4-5　方法调用

```
overloadedMethod("abc");
```

例 4-6　两个重载方法可供选择

```
private void overloadedMethod(Object o) {
    System.out.print("Object");
}

private void overloadedMethod(String s) {
    System.out.print("String");
}
```

BinaryOperator 是一种特殊的 BiFunction 类型，参数的类型和返回值的类型相同。比如，两个整数相加就是一个 BinaryOperator。

Lambda 表达式的类型就是对应的函数接口类型，因此，将 Lambda 表达式作为参数传递时，情况也依然如此。操作时可以重载一个方法，分别接受 `BinaryOperator` 和该接口的一个子类作为参数。调用这些方法时，Java 推导出的 Lambda 表达式的类型正是最具体的函数接口的类型。比如，例 4-7 在例 4-8 的两个方法中选择时，输出的是 `IntegerBinaryOperator`。

例 4-7　另外一个重载方法调用

```
overloadedMethod((x, y) -> x + y);
```

例 4-8　两个重载方法可供选择

```
private interface IntegerBiFunction extends BinaryOperator<Integer> {

}

private void overloadedMethod(BinaryOperator<Integer> Lambda) {
    System.out.print("BinaryOperator");
}
private void overloadedMethod(IntegerBiFunction Lambda) {
    System.out.print("IntegerBinaryOperator");
}
```

当然，同时存在多个重载方法时，哪个是“最具体的类型”可能并不明确。如例 4-9 所示。

例 4-9　重载方法导致的编译错误

```
overloadedMethod((x) -> true);

private interface IntPredicate {
    public boolean test(int value);
}

private void overloadedMethod(Predicate<Integer> predicate) {
    System.out.print("Predicate");
}

private void overloadedMethod(IntPredicate predicate) {
    System.out.print("IntPredicate");
}
```

传入 `overloadedMethod` 方法的 Lambda 表达式和两个函数接口 `Predicate`、`IntPredicate` 在类型上都是匹配的。在这段代码块中，两种情况都定义了相应的重载方法，这时，`javac` 就无法编译，在错误报告中显示 Lambda 表达式被模糊调用。`IntPredicate` 没有继承 `Predicate`，因此编译器无法推断出哪个类型更具体。

将 Lambda 表达式强制转换为 `IntPredicate` 或 `Predicate<Integer>` 类型可以解决这个问题，至于转换为哪种类型则取决于要调用哪个函数接口。当然，如果以前你曾自行设计过类库，就可以将其视为“代码异味”，不该再重载，而应当开始重新命名重载方法。

总而言之，Lambda 表达式作为参数时，其类型由它的目标类型推导得出，推导过程遵循如下规则：

- 如果只有一个可能的目标类型，由相应函数接口里的参数类型推导得出；
- 如果有多个可能的目标类型，由最具体的类型推导得出；
- 如果有多个可能的目标类型且最具体的类型不明确，则需人为指定类型。

4.4 @FunctionalInterface

2.4 节虽已讨论过函数接口定义的标准，但未提及 `@FunctionalInterface` 注释。事实上，每个用作函数接口的接口都应该添加这个注释。

这究竟是什么意思呢？Java 中有一些接口，虽然只含一个方法，但并不是为了使用 Lambda 表达式来实现的。比如，有些对象内部可能保存着某种状态，使用带有一个方法的接口可能纯属巧合。`java.lang.Comparable` 和 `java.io.Closeable` 就属于这样的情况。

如果一个类是可比较的，就意味着在该类的实例之间存在某种顺序，比如字符串中的字母顺序。人们通常不会认为函数是可比较的，如果一个东西既没有属性也没有状态，拿什么比较呢？

一个可关闭的对象必须持有某种打开的资源，比如一个需要关闭的文件句柄。同样，该接口也不能是一个纯函数，因为关闭资源是更改状态的另一种形式。

和 `Closeable` 和 `Comparable` 接口不同，为了提高 `Stream` 对象可操作性而引入的各种新接口，都需要有 Lambda 表达式可以实现它。它们存在的意义在于将代码块作为数据打包起来。因此，它们都添加了 `@FunctionalInterface` 注释。

该注释会强制 `javac` 检查一个接口是否符合函数接口的标准。如果该注释添加给一个枚举类型、类或另一个注释，或者接口包含不止一个抽象方法，`javac` 就会报错。重构代码时，使用它能很容易发现问题。

4.5 二进制接口的兼容性

如第 3 章开篇所言，Java 8 中对 API 最大的改变在于集合类。虽然 Java 在持续演进，但它一直在保持着向后二进制兼容。具体来说，使用 Java 1 到 Java 7 编译的类库或应用，可以直接在 Java 8 上运行。

当然，错误也难免会时有发生，但和其他编程平台相比，二进制兼容性一直被视为 Java 的关键优势所在。除非引入新的关键字，如 `enum`，达成源代码向后兼容也不是没有可能实现。可以保证，只要是 Java 1 到 Java 7 写出的代码，在 Java 8 中依然可以编译通过。

事实上，修改了像集合类这样的核心类库之后，这一保证也很难实现。我们可以用具体的例子作为思考练习。Java 8 中为 `Collection` 接口增加了 `stream` 方法，这意味着所有实现了 `Collection` 接口的类都必须增加这个新方法。对核心类库里的类来说，实现这个新方法（比如为 `ArrayList` 增加新的 `stream` 方法）就能就能使问题迎刃而解。

缺憾在于，这个修改依然打破了二进制兼容性，在 JDK 之外实现 `Collection` 接口的类，例如 `MyCustomList`，也仍然需要实现新增的 `stream` 方法。这个 `MyCustomList` 在 Java 8 中无法通过编译，即使已有一个编译好的版本，在 JVM 加载 `MyCustomList` 类时，类加载器仍然会引发异常。

这是所有使用第三方集合类库的梦魇，要避免这个糟糕情况，则需要在 Java 8 中添加新的语言特性：*默认方法*

4.6 默认方法

`Collection` 接口中增加了新的 `stream` 方法，如何能让 `MyCustomList` 类在不知道该方法的情况下通过编译？ Java 8 通过如下方法解决该问题：`Collection` 接口告诉它所有的子类："如果你没有实现 `stream` 方法，就使用我的吧。"接口中这样的方法叫作*默认方法*，在任何接口中，无论函数接口还是非函数接口，都可以使用该方法。

`Iterable` 接口中也新增了一个默认方法：`forEach`，该方法功能和 `for` 循环类似，但是允许用户使用一个 Lambda 表达式作为循环体。例 4-10 展示了 JDK 中 `forEach` 的实现方式：

例 4-10　默认方法示例：`forEach` 实现方式

```
default void forEach(Consumer<? super T> action) {
    for (T t : this) {
        action.accept(t);
    }
}
```

如果已经习惯了通过调用接口方法来使用 Lambda 表达式的方式，那么这个例子理解起来就相当简单。它使用一个常规的 `for` 循环遍历 `Iterable` 对象，然后对每个值调用 `accept` 方法。

既然如此简单，为何还要单独提出来呢？重点就在于代码段前面的新关键字 `default`。这个关键字告诉 `javac` 用户真正需要的是为接口添加一个新方法。除了添加了一个新的关键字，默认方法在继承规则上和普通方法也略有区别。

和类不同，接口没有成员变量，因此默认方法只能通过调用子类的方法来修改子类本身，避免了对子类的实现做出各种假设。

默认方法和子类

默认方法的重写规则也有一些微妙之处。从最简单的情况开始来看：没有重写。在例 4-11 中，Parent 接口定义了一个默认方法 welcome，调用该方法时，发送一条信息。ParentImpl 类没有实现 welcome 方法，因此它自然继承了该默认方法。

例 4-11　Parent 接口，其中的 welcome 是一个默认方法

```
public interface Parent {

    public void message(String body);

    public default void welcome() {
        message("Parent: Hi!");
    }

    public String getLastMessage();

}
```

在例 4-12 中调用代码，我们调用默认方法，可以看到断言正确。

例 4-12　在客户代码中使用默认方法

```
@Test
public void parentDefaultUsed() {
    Parent parent = new ParentImpl();
    parent.welcome();
    assertEquals("Parent: Hi!", parent.getLastMessage());
}
```

这时可新建一个接口 Child，继承自 Parent 接口，代码如例 4-13 所示。Child 接口实现了自己的默认 welcome 方法，凭直觉判断可知，该方法重写了 Parent 的方法。同样在这个例子中，ChildImpl 类不会实现 welcome 方法，因此它自然也继承了接口的默认方法。

例 4-13　继承了 Parent 接口的 Child 接口

```
public interface Child extends Parent {

    @Override
    public default void welcome() {
        message("Child: Hi!");
    }

}
```

此时的类继承体系如图 4-4 所示。

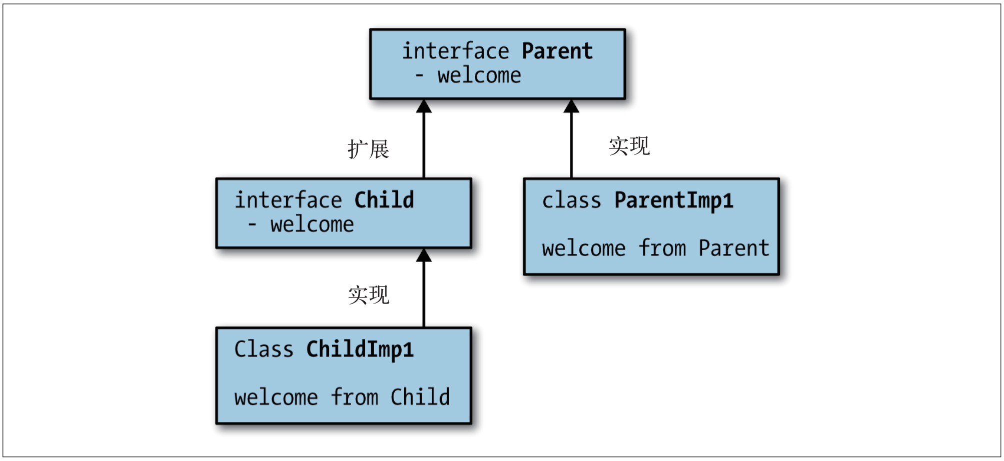


图 4-4：类继承体系图

例 4-14 调用了该接口，最后输出的字符串自然是 "Child: Hi!"。

例 4-14　调用 Child 接口的客户代码

```
@Test
public void childOverrideDefault() {
    Child child = new ChildImpl();
    child.welcome();
    assertEquals("Child: Hi!", child.getLastMessage());
}
```

现在默认方法成了虚方法——和静态方法刚好相反。任何时候，一旦与类中定义的方法产生冲突，都要优先选择类中定义的方法。例 4-15 和例 4-16 展示了这种情况，最终调用的是 OverridingParent 的，而不是 Parent 的 welcome 方法。

例 4-15　重写 welcome 默认实现的父类

```
public class OverridingParent extends ParentImpl {

    @Override
    public void welcome() {
        message("Class Parent: Hi!");
    }

}
```

例 4-16　调用的是类中的具体方法，而不是默认方法

```
@Test
public void concreteBeatsDefault() {
    Parent parent = new OverridingParent();
    parent.welcome();
    assertEquals("Class Parent: Hi!", parent.getLastMessage());
}
```

例 4-18 展示了另一种情况，或许不认为类中重写的方法能够覆盖默认方法。OverridingChild 本身并没有任何操作，只是继承了 Child 和 OverridingParent 中的 welcome 方法。最后，调用的是 OverridingParent 中的 welcome 方法，而不是 Child 接口中定义的默认方法（代码如例 4-17 所示），原因在于，与接口中定义的默认方法相比，类中重写的方法更具体（参见图 4-5）。

例 4-17　子接口重写了父接口中的默认方法

```
public class OverridingChild extends OverridingParent implements Child {

}
```

例 4-18　类中重写的方法优先级高于接口中定义的默认方法

```
@Test
public void concreteBeatsCloserDefault() {
    Child child = new OverridingChild();
    child.welcome();
    assertEquals("Class Parent: Hi!", child.getLastMessage());
}
```

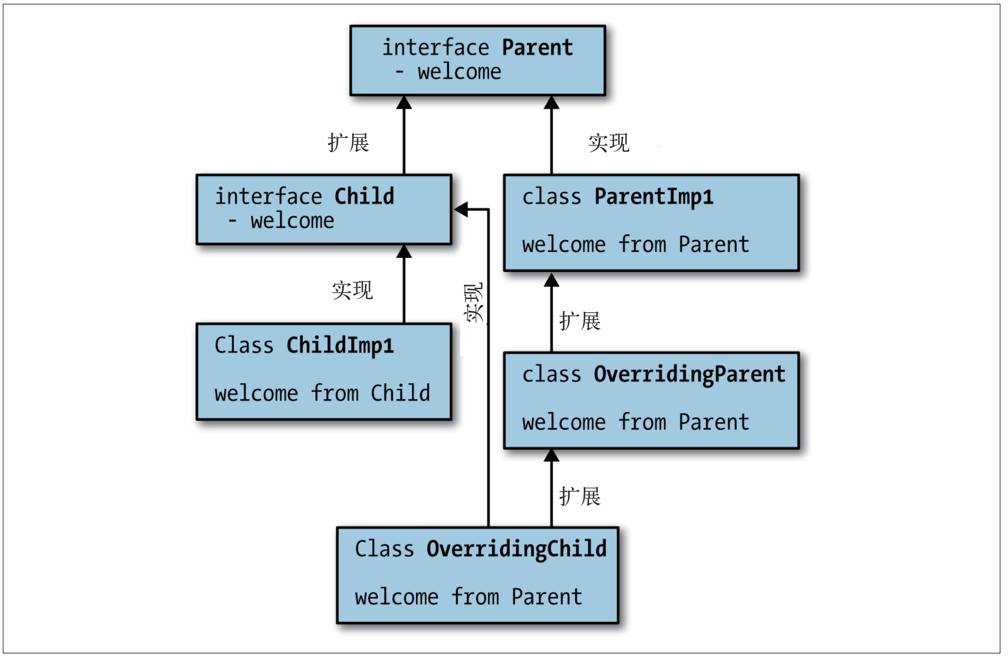


图 4-5：完整的继承体系图

简言之，*类中重写的方法胜出*。这样的设计主要是由增加默认方法的目的决定的，增加默认方法主要是为了在接口上向后兼容。让类中重写方法的优先级高于默认方法能简化很多继承问题。

假设已实现了一个定制的列表 MyCustomList，该类中有一个 addAll 方法，如果新的 List

接口也增加了一个默认方法 addAll，该方法将对列表的操作代理到 add 方法。如果类中重写的方法没有默认方法的优先级高，那么就会破坏已有的实现。

4.7 多重继承

接口允许多重继承，因此有可能碰到两个接口包含签名相同的默认方法的情况。比如例 4-19 中，接口 Carriage 和 Jukebox 都有一个默认方法 rock，虽然各有各的用途。类 MusicalCarriage 同时实现了接口 Jukebox（例 4-19）和 Carriage（例 4-20），它到底继承了哪个接口的 rock 方法呢？

例 4-19　Jukebox

```
public interface Jukebox {

    public default String rock() {
        return "... all over the world!";
    }

}
```

例 4-20　Carriage

```
public interface Carriage {

    public default String rock() {
        return "... from side to side";
    }

}

    public class MusicalCarriage implements Carriage, Jukebox {
    }
```

此时，javac 并不明确应该继承哪个接口中的方法，因此编译器会报错：class Musical Carriage inherits unrelated defaults for rock() from types Carriage and Jukebox。当然，在类中实现 rock 方法就能解决这个问题，如例 4-21 所示。

例 4-21　实现 rock 方法

```
public class MusicalCarriage
        implements Carriage, Jukebox {

    @Override
    public String rock() {
        return Carriage.super.rock();
    }

}
```

该例中使用了增强的 super 语法，用来指明使用接口 Carriage 中定义的默认方法。此前，

使用 `super` 关键字是指向父类，现在使用类似 `InterfaceName.super` 这样的语法指的是继承自父接口的方法。

三定律

如果对默认方法的工作原理，特别是在多重继承下的行为还没有把握，如下三条简单的定律可以帮助大家。

1. 类胜于接口。如果在继承链中有方法体或抽象的方法声明，那么就可以忽略接口中定义的方法。
2. 子类胜于父类。如果一个接口继承了另一个接口，且两个接口都定义了一个默认方法，那么子类中定义的方法胜出。
3. 没有规则三。如果上面两条规则不适用，子类要么需要实现该方法，要么将该方法声明为抽象方法。

其中第一条规则是为了让代码向后兼容。

4.8 权衡

在接口中定义方法的诸多变化引发了一系列问题，既然可用代码主体定义方法，那 Java 8 中的接口还是旧有版本中界定的代码吗？现在的接口提供了某种形式上的多重继承功能，然而多重继承在以前饱受诟病，Java 因此舍弃了该语言特性，这也正是 Java 在易用性方面优于 C++ 的原因之一。

语言特性的利弊也在不断演化。很多人认为多重继承的问题在于对象状态的继承，而不是代码块的继承，默认方法避免了状态的继承，也因此避免了 C++ 中多重继承的最大缺点。

突破语言上的局限性吸引着无数优秀的程序员不断尝试。现在已有一些博客文章，阐述在 Java 8 中实现完全的多重继承做出的尝试，包括状态的继承和默认方法。尝试突破 Java 8 这些有意为之的语言限制时，却往往又掉进 C++ 的旧有陷阱之中。

接口和抽象类之间还是存在明显的区别。接口允许多重继承，却没有成员变量；抽象类可以继承成员变量，却不能多重继承。在对问题域建模时，需要根据具体情况进行权衡，而在以前的 Java 中可能并不需要这样。

4.9 接口的静态方法

前面已多次出现过 `Stream.of` 方法的调用，接下来将对其进行详细介绍。`Stream` 是个接口，`Stream.of` 是接口的静态方法。这也是 Java 8 中添加的一个新的语言特性，旨在帮助编写类库的开发人员，但对于日常应用程序的开发人员也同样适用。

人们在编程过程中积累了这样一条经验，那就是一个包含很多静态方法的类。有时，类是一个放置工具方法的好地方，比如 Java 7 中引入的 `Objects` 类，就包含了很多工具方法，这些方法不是具体属于某个类的。

当然，如果一个方法有充分的语义原因和某个概念相关，那么就应该将该方法和相关的类或接口放在一起，而不是放到另一个工具类中。这有助于更好地组织代码，阅读代码的人也更容易找到相关方法。

比如，如果想创建一个由简单值组成的 `Stream`，自然希望 `Stream` 中能有一个这样的方法。这在以前很难达成，引入重接口的 `Stream` 对象，最后促使 Java 为接口加入了静态方法。

> `Stream` 和其他几个子类还包含另外几个静态方法。特别是 `range` 和 `iterate` 方法提供了产生 Stream 的其他方式。

4.10 Optional

`reduce` 方法的一个重点尚未提及：`reduce` 方法有两种形式，一种如前面出现的需要有一个初始值，另一种变式则不需要有初始值。没有初始值的情况下，`reduce` 的第一步使用 `Stream` 中的前两个元素。有时，`reduce` 操作不存在有意义的初始值，这样做就是有意义的，此时，`reduce` 方法返回一个 `Optional` 对象。

`Optional` 是为核心类库新设计的一个数据类型，用来替换 `null` 值。人们对原有的 `null` 值有很多抱怨，甚至连发明这一概念的 Tony Hoare 也是如此，他曾说这是自己的一个“价值连城的错误”。作为一名有影响力的计算机科学家就是这样：虽然连一毛钱也见不到，却也可以犯一个“价值连城的错误”。

人们常常使用 `null` 值表示值不存在，`Optional` 对象能更好地表达这个概念。使用 `null` 代表值不存在的最大问题在于 `NullPointerException`。一旦引用一个存储 `null` 值的变量，程序会立即崩溃。使用 `Optional` 对象有两个目的：首先，`Optional` 对象鼓励程序员适时检查变量是否为空，以避免代码缺陷；其次，它将一个类的 API 中可能为空的值文档化，这比阅读实现代码要简单很多。

下面我们举例说明 `Optional` 对象的 API，从而切身体会一下它的使用方法。使用工厂方法 `of`，可以从某个值创建出一个 `Optional` 对象。`Optional` 对象相当于值的容器，而该值可以通过 `get` 方法提取。如例 4-22 所示。

例 4-22　创建某个值的 `Optional` 对象

```
Optional<String> a = Optional.of("a");
assertEquals("a", a.get());
```

Optional 对象也可能为空，因此还有一个对应的工厂方法 empty，另外一个工厂方法 ofNullable 则可将一个空值转换成 Optional 对象。例 4-23 展示了这两个方法，同时展示了第三个方法 isPresent 的用法（该方法表示一个 Optional 对象里是否有值）。

例 4-23　创建一个空的 Optional 对象，并检查其是否有值

```
Optional emptyOptional = Optional.empty();
Optional alsoEmpty = Optional.ofNullable(null);

assertFalse(emptyOptional.isPresent());

// 例 4-22 中定义了变量 a
assertTrue(a.isPresent());
```

使用 Optional 对象的方式之一是在调用 get() 方法前，先使用 isPresent 检查 Optional 对象是否有值。使用 orElse 方法则更简洁，当 Optional 对象为空时，该方法提供了一个备选值。如果计算备选值在计算上太过繁琐，即可使用 orElseGet 方法。该方法接受一个 Supplier 对象，只有在 Optional 对象真正为空时才会调用。例 4-24 展示了这两个方法。

例 4-24　使用 orElse 和 orElseGet 方法

```
assertEquals("b", emptyOptional.orElse("b"));
assertEquals("c", emptyOptional.orElseGet(() -> "c"));
```

Optional 对象不仅可以用于新的 Java 8 API，也可用于具体领域类中，和普通的类别无二致。当试图避免空值相关的缺陷，如未捕获的异常时，可以考虑一下是否可使用 Optional 对象。

4.11　要点回顾

- 使用为基本类型定制的 Lambda 表达式和 Stream，如 IntStream 可以显著提升系统性能。
- 默认方法是指接口中定义的包含方法体的方法，方法名有 default 关键字做前缀。
- 在一个值可能为空的建模情况下，使用 Optional 对象能替代使用 null 值。

4.12　练习

1. 在例 4-25 所示的 Performance 接口基础上，添加 getAllMusicians 方法，该方法返回包含所有艺术家名字的 Stream，如果对象是乐队，则返回每个乐队成员的名字。例如，如果 getMusicians 方法返回甲壳虫乐队，则 getAllMusicians 方法返回乐队名和乐队成员，如约翰 · 列侬、保罗 · 麦卡特尼等。

例 4-25　表示音乐表演的接口

```
/** 该接口表示艺术家的演出——专辑或演唱会 */
public interface Performance {
```

```
        public String getName();

        public Stream<Artist> getMusicians();

    }
```

2. 根据前面描述的重载解析规则，能否重写默认方法中的 `equals` 或 `hashCode` 方法？
3. 例 4-26 所示的 `Artists` 类表示了一组艺术家，重构该类，使得 `getArtist` 方法返回一个 `Optional<Artist>` 对象。如果索引在有效范围内，返回对应的元素，否则返回一个空 `Optional` 对象。此外，还需重构 `getArtistName` 方法，保持相同的行为。

例 4-26　包含多个艺术家的 Artists 类

```
    public class Artists {

        private List<Artist> artists;

        public Artists(List<Artist> artists) {
            this.artists = artists;
        }

        public Artist getArtist(int index) {
            if (index < 0 || index >= artists.size()) {
                indexException(index);
            }
            return artists.get(index);
        }

        private void indexException(int index) {
            throw new IllegalArgumentException(index +
                                                "doesn't correspond to an Artist");
        }

        public String getArtistName(int index) {
            try {
                Artist artist = getArtist(index);
                return artist.getName();
            } catch (IllegalArgumentException e) {
                return "unknown";
            }
        }

    }
```

4.13　开放练习

审阅工作代码库或熟悉的开源项目代码，找出哪些只包含静态方法的类适合用包含静态方法的接口替代。如有可能，和同事一起讨论，看他们是否赞同你找出的结果。

第 5 章

高级集合类和收集器

第 3 章只介绍了集合类的部分变化，事实上，Java 8 对集合类的改进不止这些。现在是时候介绍一些高级主题了，包括新引入的 Collector 类。同时我还会为大家介绍方法引用，它可以帮助大家在 Lambda 表达式中轻松使用已有代码。编写大量使用集合类的代码时，使用方法引用能让程序员获得丰厚的回报。本章还会涉及集合类的一些更高级的主题，比如流中元素的顺序，以及一些有用的 API。

5.1　方法引用

读者可能已经发现，Lambda 表达式有一个常见的用法：Lambda 表达式经常调用参数。比如想得到艺术家的姓名，Lambda 的表达式如下：

```
artist -> artist.getName()
```

这种用法如此普遍，因此 Java 8 为其提供了一个简写语法，叫作*方法引用*，帮助程序员重用已有方法。用方法引用重写上面的 Lambda 表达式，代码如下：

```
Artist::getName
```

标准语法为 Classname::methodName。需要注意的是，虽然这是一个方法，但不需要在后面加括号，因为这里并不调用该方法。我们只是提供了和 Lambda 表达式等价的一种结构，在需要时才会调用。凡是使用 Lambda 表达式的地方，就可以使用方法引用。

构造函数也有同样的缩写形式，如果你想使用 Lambda 表达式创建一个 Artist 对象，可能会写出如下代码：

```
(name, nationality) -> new Artist(name, nationality)
```

使用方法引用，上述代码可写为：

```
Artist::new
```

这段代码不仅比原来的代码短，而且更易阅读。`Artist::new` 立刻告诉程序员这是在创建一个 `Artist` 对象，程序员无需看完整行代码就能弄明白代码的意图。另一个要注意的地方是方法引用自动支持多个参数，前提是选对了正确的函数接口。

还可以用这种方式创建数组，下面的代码创建了一个字符串型的数组：

```
String[]::new
```

从现在开始，我们将在合适的地方使用方法引用，因此读者很快会看到更多的例子。一开始探索 Java 8 时，有位朋友告诉我，方法引用看起来"就像在作弊"。他的意思是说，了解如何使用 Lambda 表达式让代码像数据一样在对象间传递之后，这种直接引用方法的方式就像"作弊"。

放心，这不是在作弊。读者只要记住，每次写出形如 `x -> foo(x)` 的 Lambda 表达式时，和直接调用方法 `foo` 是一样的。方法引用只不过是基于这样的事实，提供了一种简短的语法而已。

5.2 元素顺序

另外一个尚未提及的关于集合类的内容是流中的元素以何种顺序排列。读者可能知道，一些集合类型中的元素是按顺序排列的，比如 `List`；而另一些则是无序的，比如 `HashSet`。增加了流操作后，顺序问题变得更加复杂。

直观上看，流是有序的，因为流中的元素都是按顺序处理的。这种顺序称为*出现顺序*。出现顺序的定义依赖于数据源和对流的操作。

在一个有序集合中创建一个流时，流中的元素就按出现顺序排列，因此，例 5-1 中的代码总是可以通过。

例 5-1 顺序测试永远通过

```
List<Integer> numbers = asList(1, 2, 3, 4);

List<Integer> sameOrder = numbers.stream()
                                 .collect(toList());
assertEquals(numbers, sameOrder);
```

如果集合本身就是无序的，由此生成的流也是无序的。`HashSet` 就是一种无序的集合，因此不能保证例 5-2 所示的程序每次都通过。

例 5-2　顺序测试不能保证每次通过

```
Set<Integer> numbers = new HashSet<>(asList(4, 3, 2, 1));

List<Integer> sameOrder = numbers.stream()
                                 .collect(toList());

// 该断言有时会失败
assertEquals(asList(4, 3, 2, 1), sameOrder);
```

流的目的不仅是在集合类之间做转换，而且同时提供了一组处理数据的通用操作。有些集合本身是无序的，但这些操作有时会产生顺序，试看例 5-3 中的代码。

例 5-3　生成出现顺序

```
Set<Integer> numbers = new HashSet<>(asList(4, 3, 2, 1));

List<Integer> sameOrder = numbers.stream()
                                 .sorted()
                                 .collect(toList());

assertEquals(asList(1, 2, 3, 4), sameOrder);
```

一些中间操作会产生顺序，比如对值做映射时，映射后的值是有序的，这种顺序就会保留下来。如果进来的流是无序的，出去的流也是无序的。看一下例 5-4 所示代码，我们只能断言 HashSet 中含有某元素，但对其顺序不能作出任何假设，因为 HashSet 是无序的，使用了映射操作后，得到的集合仍然是无序的。

例 5-4　本例中关于顺序的假设永远是正确的

```
List<Integer> numbers = asList(1, 2, 3, 4);

List<Integer> stillOrdered = numbers.stream()
                                    .map(x -> x + 1)
                                    .collect(toList());

// 顺序得到了保留
assertEquals(asList(2, 3, 4, 5), stillOrdered);

Set<Integer> unordered = new HashSet<>(numbers);

List<Integer> stillUnordered = unordered.stream()
                                        .map(x -> x + 1)
                                        .collect(toList());

// 顺序得不到保证
assertThat(stillUnordered, hasItem(2));
assertThat(stillUnordered, hasItem(3));
assertThat(stillUnordered, hasItem(4));
assertThat(stillUnordered, hasItem(5));
```

一些操作在有序的流上开销更大，调用 unordered 方法消除这种顺序就能解决该问题。大多数操作都是在有序流上效率更高，比如 filter、map 和 reduce 等。

这会带来一些意想不到的结果，比如使用并行流时，`forEach` 方法不能保证元素是按顺序处理的（第 6 章会详细讨论这些内容）。如果需要保证按顺序处理，应该使用 `forEachOrdered` 方法，它是你的朋友。

5.3 使用收集器

前面我们使用过 `collect(toList())`，在流中生成列表。显然，`List` 是能想到的从流中生成的最自然的数据结构，但是有时人们还希望从流生成其他值，比如 `Map` 或 `Set`，或者你希望定制一个类将你想要的东西抽象出来。

前面已经讲过，仅凭流上方法的签名，就能判断出这是否是一个及早求值的操作。`reduce` 操作就是一个很好的例子，但有时人们希望能做得更多。

这就是*收集器*，一种通用的、从流生成复杂值的结构。只要将它传给 `collect` 方法，所有的流就都可以使用它了。

标准类库已经提供了一些有用的收集器，让我们先来看看。本章示例代码中的收集器都是从 `java.util.stream.Collectors` 类中静态导入的。

5.3.1 转换成其他集合

有一些收集器可以生成其他集合。比如前面已经见过的 `toList`，生成了 `java.util.List` 类的实例。还有 `toSet` 和 `toCollection`，分别生成 `Set` 和 `Collection` 类的实例。到目前为止，我已经讲了很多流上的链式操作，但总有一些时候，需要最终生成一个集合——比如：

- 已有代码是为集合编写的，因此需要将流转换成集合传入；
- 在集合上进行一系列链式操作后，最终希望生成一个值；
- 写单元测试时，需要对某个具体的集合做断言。

通常情况下，创建集合时需要调用适当的构造函数指明集合的具体类型：

```
List<Artist> artists = new ArrayList<>();
```

但是调用 `toList` 或者 `toSet` 方法时，不需要指定具体的类型。`Stream` 类库在背后自动为你挑选出了合适的类型。本书后面会讲述如何使用 `Stream` 类库并行处理数据，收集并行操作的结果需要的 `Set`，和对线程安全没有要求的 `Set` 类是完全不同的。

可能还会有这样的情况，你希望使用一个特定的集合收集值，而且你可以稍后指定该集合的类型。比如，你可能希望使用 `TreeSet`，而不是由框架在背后自动为你指定一种类型的 `Set`。此时就可以使用 `toCollection`，它接受一个函数作为参数，来创建集合（见例 5-5）。

例 5-5　使用 toCollection，用定制的集合收集元素

```
stream.collect(toCollection(TreeSet::new));
```

5.3.2　转换成值

还可以利用收集器让流生成一个值。maxBy 和 minBy 允许用户按某种特定的顺序生成一个值。例 5-6 展示了如何找出成员最多的乐队。它使用一个 Lambda 表达式，将艺术家映射为成员数量，然后定义了一个比较器，并将比较器传入 maxBy 收集器。

例 5-6　找出成员最多的乐队

```
public Optional<Artist> biggestGroup(Stream<Artist> artists) {
    Function<Artist,Long> getCount = artist -> artist.getMembers().count();
    return artists.collect(maxBy(comparing(getCount)));
}
```

minBy 就如它的方法名，是用来找出最小值的。

还有些收集器实现了一些常用的数值运算。让我们通过一个计算专辑曲目平均数的例子来看看，如例 5-7 所示。

例 5-7　找出一组专辑上曲目的平均数


```
public double averageNumberOfTracks(List<Album> albums) {
    return albums.stream()
                 .collect(averagingInt(album -> album.getTrackList().size()));
}
```

和以前一样，通过调用 stream 方法让集合生成流，然后调用 collect 方法收集结果。averagingInt 方法接受一个 Lambda 表达式作参数，将流中的元素转换成一个整数，然后再计算平均数。还有和 double 和 long 类型对应的重载方法，帮助程序员将元素转换成相应类型的值。

第 4 章介绍过一些特殊的流，如 IntStream，为数值运算定义了一些额外的方法。事实上，Java 8 也提供了能完成类似功能的收集器，如 averagingInt。可以使用 summingInt 及其重载方法求和。SummaryStatistics 也可以使用 summingInt 及其组合收集。

5.3.3　数据分块

另外一个常用的流操作是将其分解成两个集合。假设有一个艺术家组成的流，你可能希望将其分成两个部分，一部分是独唱歌手，另一部分是由多人组成的乐队。可以使用两次过滤操作，分别过滤出上述两种艺术家。

但是这样操作起来有问题。首先，为了执行两次过滤操作，需要有两个流。其次，如果过滤操作复杂，每个流上都要执行这样的操作，代码也会变得冗余。

幸好我们有这样一个收集器 partitioningBy，它接受一个流，并将其分成两部分（如图

5-1 所示）。它使用 Predicate 对象判断一个元素应该属于哪个部分，并根据布尔值返回一个 Map 到列表。因此，对于 true List 中的元素，Predicate 返回 true；对其他 List 中的元素，Predicate 返回 false。

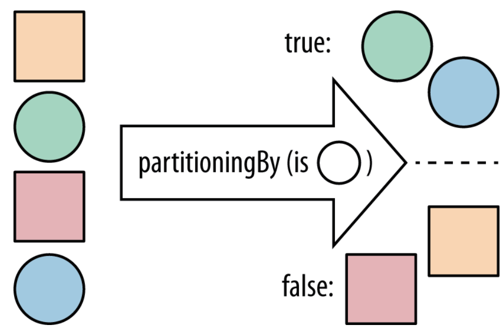


图 5-1：partitioningBy 收集器

使用它，我们就可以将乐队（有多个成员）和独唱歌手分开了。在本例中，分块函数指明艺术家是否为独唱歌手。实现如例 5-8 所示。

例 5-8　将艺术家组成的流分成乐队和独唱歌手两部分

```
public Map<Boolean, List<Artist>> bandsAndSolo(Stream<Artist> artists) {
    return artists.collect(partitioningBy(artist -> artist.isSolo()));
}
```

也可以使用方法引用代替 Lambda 表达式，如例 5-9 所示。

例 5-9　使用方法引用将艺术家组成的 Stream 分成乐队和独唱歌手两部分

```
public Map<Boolean, List<Artist>> bandsAndSoloRef(Stream<Artist> artists) {
    return artists.collect(partitioningBy(Artist::isSolo));
}
```

5.3.4　数据分组

数据分组是一种更自然的分割数据操作，与将数据分成 ture 和 false 两部分不同，可以使用任意值对数据分组。比如现在有一个由专辑组成的流，可以按专辑当中的主唱对专辑分组。代码如例 5-10 所示。

例 5-10　使用主唱对专辑分组

```
public Map<Artist, List<Album>> albumsByArtist(Stream<Album> albums) {
    return albums.collect(groupingBy(album -> album.getMainMusician()));
}
```

和其他例子一样，调用流的 collect 方法，传入一个收集器。groupingBy 收集器（如图 5-2 所示）接受一个分类函数，用来对数据分组，就像 partitioningBy 一样，接受一个 Predicate 对象将数据分成 ture 和 false 两部分。我们使用的分类器是一个 Function 对象，和 map 操作用到的一样。

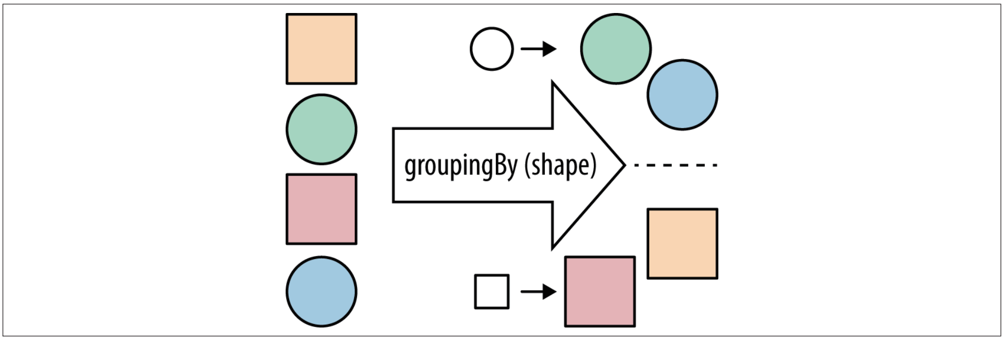
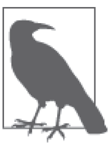


图 5-2：groupingBy 收集器



读者可能知道 SQL 中的 group by 操作，我们的方法是和这类似的一个概念，只不过在 Stream 类库中实现了而已。

5.3.5 字符串

很多时候，收集流中的数据都是为了在最后生成一个字符串。假设我们想将参与制作一张专辑的所有艺术家的名字输出为一个格式化好的列表，以专辑 *Let It Be* 为例，期望的输出为："[George Harrison, John Lennon, Paul McCartney, Ringo Starr, The Beatles]"。

在 Java 8 还未发布前，实现该功能的代码可能如例 5-11 所示。通过不断迭代列表，使用一个 StringBuilder 对象来记录结果。每一步都取出一个艺术家的名字，追加到 StringBuilder 对象。

例 5-11　使用 for 循环格式化艺术家姓名

```
StringBuilder builder = new StringBuilder("[");
for (Artist artist : artists) {
    if (builder.length() > 1)
        builder.append(", ");

    String name = artist.getName();
    builder.append(name);
}
builder.append("]");
String result = builder.toString();
```

显然，这段代码不是非常好。如果不一步步跟踪，很难看出这段代码是干什么的。使用 Java 8 提供的流和收集器就能写出更清晰的代码，如例 5-12 所示。

例 5-12　使用流和收集器格式化艺术家姓名

```
String result =
    artists.stream()
              .map(Artist::getName)
              .collect(Collectors.joining(", ", "[", "]"));
```

这里使用 `map` 操作提取出艺术家的姓名，然后使用 `Collectors.joining` 收集流中的值，该方法可以方便地从一个流得到一个字符串，允许用户提供分隔符（用以分隔元素）、前缀和后缀。

5.3.6　组合收集器

虽然读者现在看到的各种收集器已经很强大了，但如果将它们组合起来，会变得更强大。

之前我们使用主唱将专辑分组，现在来考虑如何计算一个艺术家的专辑数量。一个简单的方案是使用前面的方法对专辑先分组后计数，如例 5-13 所示。

例 5-13　计算每个艺术家专辑数的简单方式

```
Map<Artist, List<Album>> albumsByArtist
    = albums.collect(groupingBy(album -> album.getMainMusician()));

Map<Artist, Integer> numberOfAlbums = new HashMap<>();
for(Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
    numberOfAlbums.put(entry.getKey(), entry.getValue().size());
}
```

这种方式看起来简单，但却有点杂乱无章。这段代码也是命令式的代码，不能自动适应并行化操作。

这里实际上需要另外一个收集器，告诉 `groupingBy` 不用为每一个艺术家生成一个专辑列表，只需要对专辑计数就可以了。幸好，核心类库已经提供了一个这样的收集器：`counting`。使用它，可将上述代码重写为例 5-14 所示的样子。

例 5-14　使用收集器计算每个艺术家的专辑数

```
public Map<Artist, Long> numberOfAlbums(Stream<Album> albums) {
    return albums.collect(groupingBy(album -> album.getMainMusician(),
                                     counting()));
}
```

`groupingBy` 先将元素分成块，每块都与分类函数 `getMainMusician` 提供的键值相关联，然后使用下游的另一个收集器收集每块中的元素，最好将结果映射为一个 `Map`。

让我们再看一个例子，这次我们不想生成一组专辑，只希望得到专辑名。这个问题仍然可以用前面的方法解决，先将专辑分组，然后再调整生成的 `Map` 中的值，如例 5-15 所示。

例 5-15　使用简单方式求每个艺术家的专辑名

```
public Map<Artist, List<String>> nameOfAlbumsDumb(Stream<Album> albums) {
    Map<Artist, List<Album>> albumsByArtist =
            albums.collect(groupingBy(album ->album.getMainMusician()));

    Map<Artist, List<String>> nameOfAlbums = new HashMap<>();
    for(Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
        nameOfAlbums.put(entry.getKey(), entry.getValue()
                                              .stream()
                                              .map(Album::getName)
                                              .collect(toList()));
    }
    return nameOfAlbums;
}
```

同理，我们可以再使用一个收集器，编写出更好、更快、更容易并行处理的代码。我们已经知道，可以使用 `groupingBy` 将专辑按主唱分组，但是其输出为一个 `Map<Artist, List<Album>>` 对象，它将每个艺术家和他的专辑列表关联起来，但这不是我们想要的，我们想要的是一个包含专辑名的字符串列表。

此时，我们真正想做的是将专辑列表映射为专辑名列表，这里不能直接使用流的 `map` 操作，因为列表是由 `groupingBy` 生成的。我们需要有一种方法，可以告诉 `groupingBy` 将它的值做映射，生成最终结果。

每个收集器都是生成最终值的一剂良方。这里需要两剂配方，一个传给*另*一个。谢天谢地，Oracle 公司的研究员们已经考虑到这种情况，为我们提供了 `mapping` 收集器。

`mapping` 允许在收集器的容器上执行类似 `map` 的操作。但是需要指明使用什么样的集合类存储结果，比如 `toList`。这些收集器就像乌龟叠罗汉，龟龟相驮以至无穷。

`mapping` 收集器和 `map` 方法一样，接受一个 `Function` 对象作为参数，经过重构后的代码如例 5-16 所示。

例 5-16　使用收集器求每个艺术家的专辑名

```
public Map<Artist, List<String>> nameOfAlbums(Stream<Album> albums) {
    return albums.collect(groupingBy(Album::getMainMusician,
                                     mapping(Album::getName, toList())));
}
```

这两个例子中我们都用到了第二个收集器，用以收集最终结果的一个子集。这些收集器叫作*下游收集器*。收集器是生成最终结果的一剂配方，下游收集器则是生成部分结果的配方，主收集器中会用到下游收集器。这种组合使用收集器的方式，使得它们在 `Stream` 类库中的作用更加强大。

那些为基本类型特殊定制的函数，如 `averagingInt`、`summarizingLong` 等，事实上和调用特殊 `Stream` 上的方法是等价的，加上它们是为了将它们当作下游收集器来使用的。

5.3.7　重构和定制收集器

尽管在常用流操作里，Java 内置的收集器已经相当好用，但收集器框架本身是极其通用的。JDK 提供的收集器没有什么特别的，完全可以定制自己的收集器，而且定制起来相当简单，这就是本节要讲的内容。

读者可能还没忘记在例 5-11 中，如何使用 Java 7 连接字符串，尽管形式并不优雅。让我们逐步重构这段代码，最终用合适的收集器实现原有代码功能。在工作中没有必要这样做，JDK 已经提供了一个完美的收集器 `joining`。这里只是为了展示如何定制收集器，以及如何使用 Java 8 提供的新功能来重构遗留代码。

例 5-17 复制了例 5-11，展示了如何在 Java 7 中连接字符串。

例 5-17　使用 `for` 循环和 `StringBuilder` 格式化艺术家姓名

```
StringBuilder builder = new StringBuilder("[");
for (Artist artist : artists) {
    if (builder.length() > 1)
        builder.append(", ");
    String name = artist.getName();
    builder.append(name);
}
builder.append("]");
String result = builder.toString();
```

显然，可以使用 `map` 操作，将包含艺术家的流映射为包含艺术家姓名的流。例 5-18 展示了使用了流的 `map` 操作重构后的代码。

例 5-18　使用 `forEach` 和 `StringBuilder` 格式化艺术家姓名

```
StringBuilder builder = new StringBuilder("[");
artists.stream()
       .map(Artist::getName)
       .forEach(name -> {
           if (builder.length() > 1)
               builder.append(", ");
           builder.append(name);
       });
builder.append("]");
String result = builder.toString();
```

将艺术家映射为姓名，就能更快看出最终是要生成什么，这样代码看起来更清楚一点。可惜 `forEach` 方法看起来还是有点笨重，这与我们通过组合高级操作让代码变得易读的目标不符。

暂且不必考虑定制一个收集器，让我们想想怎么通过流上已有的操作来解决该问题。和生成字符串目标最近的操作就是 `reduce`，使用它将例 5-18 中的代码重构如下。

例 5-19　使用 reduce 和 StringBuilder 格式化艺术家姓名

```
StringBuilder reduced =
    artists.stream()
           .map(Artist::getName)
           .reduce(new StringBuilder(), (builder, name) -> {
                   if (builder.length() > 0)
                       builder.append(", ");

                   builder.append(name);
                   return builder;
               }, (left, right) -> left.append(right));

reduced.insert(0, "[");
reduced.append("]");
String result = reduced.toString();
```

我曾经天真地以为上面的重构会让代码变得更清晰，可惜恰好相反，代码看起来比以前更糟糕。让我们先来看看怎么回事。和前面的例子一样，都调用了 stream 和 map 方法，reduce 操作生成艺术家姓名列表，艺术家与艺术家之间用“,”分隔。首先创建一个 StringBuilder 对象，该对象是 reduce 操作的初始状态，然后使用 Lambda 表达式将姓名连接到 builder 上。reduce 操作的第三个参数也是一个 Lambda 表达式，接受两个 StringBuilder 对象做参数，将两者连接起来。最后添加前缀和后缀。

在接下来的重构中，我们还是使用 reduce 操作，不过需要将杂乱无章的代码隐藏掉——我的意思是使用一个 StringCombiner 类对细节进行抽象。代码如例 5-20 所示。

例 5-20　使用 reduce 和 StringCombiner 类格式化艺术家姓名

```
StringCombiner combined =
        artists.stream()
               .map(Artist::getName)
               .reduce(new StringCombiner(", ", "[", "]"),
                       StringCombiner::add,
                       StringCombiner::merge);
String result = combined.toString();
```

尽管代码看起来和上个例子大相径庭，其实背后做的工作是一样的。我们使用 reduce 操作将姓名和分隔符连接成一个 StringBuilder 对象。不过这次连接姓名操作被代理到了 StringCombiner.add 方法，而连接两个连接器操作被 StringCombiner.merge 方法代理。让我们现在来看看这些方法，先从例 5-21 中的 add 方法开始。

例 5-21　add 方法返回连接新元素后的结果

```
public StringCombiner add(String element) {
    if (areAtStart()) {
        builder.append(prefix);
    } else {
        builder.append(delim);
    }
    builder.append(element);
```

```
        return this;
    }
```

add 方法在内部其实将操作代理给一个 StringBuilder 对象。如果刚开始进行连接，则在最前面添加前缀，否则添加分隔符，然后再添加新的元素。这里返回一个 StringCombiner 对象，因为这是传给 reduce 操作所需要的类型。合并代码也是同样的道理，内部将操作代理给 StringBuilder 对象，如例 5-22 所示。

例 5-22　merge 方法连接两个 StringCombiner 对象

```
    public StringCombiner merge(StringCombiner other) {
        builder.append(other.builder);
        return this;
    }
```

reduce 阶段的重构还差一小步就差不多结束了。我们要在最后调用 toString 方法，将整个步骤串成一个方法链。这很简单，只需要排列好 reduce 代码，准备好将其转换为 Collector API 就行了（如例 5-23 所示）。

例 5-23　使用 reduce 操作，将工作代理给 StringCombiner 对象

```
    String result =
        artists.stream()
                .map(Artist::getName)
                .reduce(new StringCombiner(", ", "[", "]"),
                        StringCombiner::add,
                        StringCombiner::merge)
                .toString();
```

现在的代码看起来已经差不多完美了，但是在程序中还是不能重用。因此，我们想将 reduce 操作重构为一个收集器，在程序中的任何地方都能使用。不妨将这个收集器叫作 StringCollector，让我们重构代码使用这个新的收集器，如例 5-24 所示。

例 5-24　使用定制的收集器 StringCollector 收集字符串

```
    String result =
        artists.stream()
                .map(Artist::getName)
                .collect(new StringCollector(", ", "[", "]"));
```

既然已经将所有对字符串的连接操作代理给了定制的收集器，应用程序就不需要关心 StringCollector 对象的任何内部细节，它和框架中其他 Collector 对象用起来是一样的。

先来实现 Collector 接口（例 5-25），由于 Collector 接口支持泛型，因此先得确定一些具体的类型：

- 待收集元素的类型，这里是 String；
- 累加器的类型 StringCombiner；
- 最终结果的类型，这里依然是 String。

例 5-25　定义字符串收集器

```
public class StringCollector implements Collector<String, StringCombiner, String> {
```

一个收集器由四部分组成。首先是一个 Supplier，这是一个工厂方法，用来创建容器，在这个例子中，就是 StringCombiner。和 reduce 操作中的第一个参数类似，它是后续操作的初值（如例 5-26 所示）。

例 5-26　Supplier 是创建容器的工厂

```
public Supplier<StringCombiner> supplier() {
    return () -> new StringCombiner(delim, prefix, suffix);
}
```

让我们一边阅读代码，一边看图，这样就能看清到底是怎么工作的。由于收集器可以并行收集，我们要展示的收集操作在两个容器上（比如 StringCombiners）并行进行。

收集器的每一个组件都是函数，因此我们使用箭头表示，流中的值用圆圈表示，最终生成的值用椭圆表示。收集操作一开始，Supplier 先创建出新的容器（如图 5-3）。

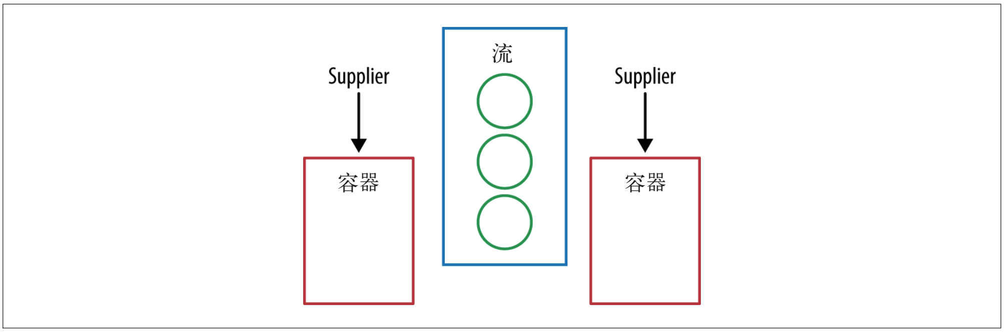


图 5-3：Supplier

收集器的 accumulator 的作用和 reduce 操作的第二个参数一样，它结合之前操作的结果和当前值，生成并返回新的值。这一逻辑已经在 StringCombiners 的 add 方法中得以实现，直接引用就好了（如例 5-27 所示）。

例 5-27　accumulator 是一个函数，它将当前元素叠加到收集器

```
public BiConsumer<StringCombiner, String> accumulator() {
    return StringCombiner::add;
}
```

这里的 accumulator 用来将流中的值叠加入容器中（如图 5-4 所示）。

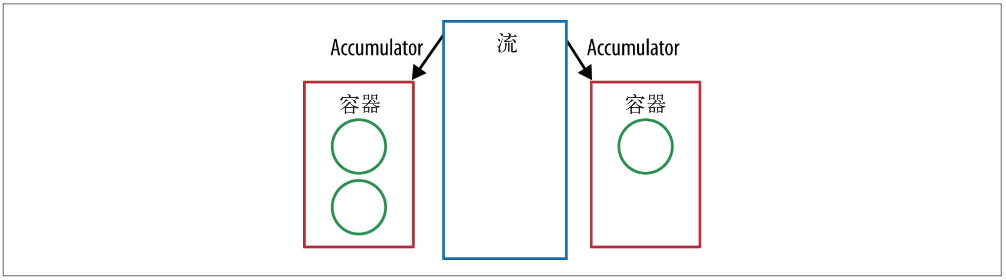


图 5-4：Accumulator

combine 方法很像 reduce 操作的第三个方法。如果有两个容器，我们需要将其合并。同样，在前面的重构中我们已经实现了该功能，直接使用 StringCombiner.merge 方法就行了（例 5-28）。

例 5-28　combiner 合并两个容器

```
public BinaryOperator<StringCombiner> combiner() {
    return StringCombiner::merge;
}
```

在收集阶段，容器被 combiner 方法成对合并进一个容器，直到最后只剩一个容器为止（如图 5-5 所示）。

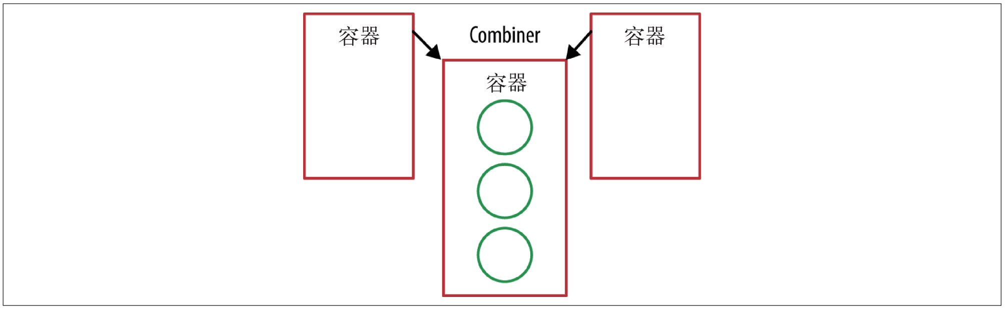


图 5-5：Combiner

读者可能还记得，在使用收集器之前，重构的最后一步将 toString 方法内联到方法链的末端，这就将 StringCombiners 转换成了我们想要的字符串（如图 5-6 所示）。

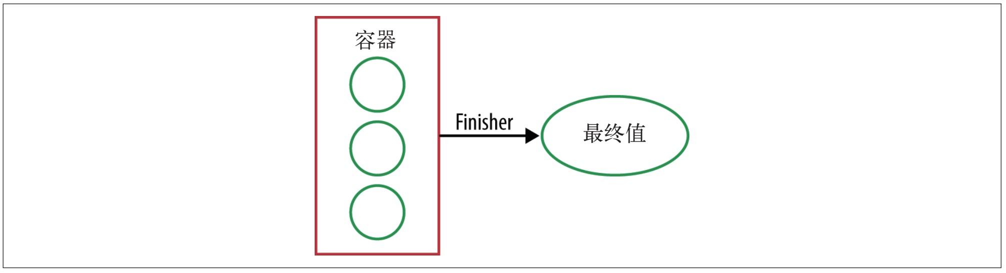


图 5-6：Finisher

收集器的 finisher 方法作用相同。我们已经将流中的值叠加入一个可变容器中，但这还不是我们想要的最终结果。这里调用了 finisher 方法，以便进行转换。在我们想创建字符串等不可变的值时特别有用，这里容器是可变的。

为了实现 finisher 方法，只需将该操作代理给已经实现的 toString 方法即可（例 5-29）。

例 5-29　finisher 方法返回收集操作的最终结果

```
public Function<StringCombiner, String> finisher() {
    return StringCombiner::toString;
}
```

从最后剩下的容器中得到最终结果。

关于收集器，还有一点一直没有提及，那就是特征。特征是一组描述收集器的对象，框架可以对其适当优化。characteristics 方法定义了特征。

在这里我有必要重申，这些代码只作教学用途，和 joining 收集器的内部实现略有出入。读者也许会认为 StringCombiner 看起来非常有用，别担心——你没必要亲自去编写，Java 8 有一个 java.util.StringJoiner 类，它的作用和 StringCombiner 一样，有类似的 API。

做这些练习的主要目的不仅在于展示定制收集器的工作原理，而且还在于帮助读者编写自己的收集器。特别是你有自己特定领域内的类，希望从集合中构建一个操作，而标准的集合类并没有提供这种操作时，就需要定制自己的收集器。

以 StringCombiner 为例，收集值的容器和我们想要创建的值（字符串）不一样。如果想要收集的是不可变对象，而不是可变对象，那么这种情况就非常普遍，否则收集操作的每一步都需要创建一个新值。

想要收集的最终结果和容器一样是完全有可能的。事实上，如果收集的最终结果是集合，比如 toList 收集器，就属于这种情况。

此时，finisher 方法不需要对容器做任何操作。更正式地说，此时的 finisher 方法其实是 identity 函数：它返回传入参数的值。如果这样，收集器就展现出 IDENTITY_FINISH 的特征，需要使用 characteristics 方法声明。

5.3.8　对收集器的归一化处理

就像之前看到的那样，定制收集器其实不难，但如果你想为自己领域内的类定制一个收集器，不妨考虑一下其他替代方案。最容易想到的方案是构建若干个集合对象，作为参数传给领域内类的构造函数。如果领域内的类包含多种集合，这种方式又简单又适用。

当然，如果领域内的类没有这些集合，需要在已有数据上计算，那这种方法就不合适了。但即使如此，也不见得需要定制一个收集器。你还可以使用 reducing 收集器，它为流上的

归一操作提供了统一实现。例 5-30 展示了如何使用 reducing 收集器编写字符串处理程序。

例 5-30　reducing 是一种定制收集器的简便方式

```
String result =
        artists.stream()
               .map(Artist::getName)
               .collect(Collectors.reducing(
                   new StringCombiner(", ", "[", "]"),
                   name -> new StringCombiner(", ", "[", "]").add(name),
                   StringCombiner::merge))
               .toString();
```

这和我在例 5-20 中讲到的基于 reduce 操作的实现很像，这点从方法名中就能看出。区别在于 Collectors.reducing 的第二个参数，我们为流中每个元素创建了唯一的 StringCombiner。如果你被这种写法吓到了，或是感到恶心，你不是一个人！这种方式非常低效，这也是我要定制收集器的原因之一。

5.4　一些细节

Lambda 表达式的引入也推动了一些新方法被加入集合类。让我们来看看 Map 类的一些变化。

构建 Map 时，为给定值计算键值是常用的操作之一，一个经典的例子就是实现一个缓存。传统的处理方式是先试着从 Map 中取值，如果没有取到，创建一个新值并返回。

假设使用 Map<String, Artist> artistCache 定义缓存，我们需要使用费时的数据库操作查询艺术家信息，代码可能如例 5-31 所示。

例 5-31　使用显式判断空值的方式缓存

```
public Artist getArtist(String name) {
    Artist artist = artistCache.get(name);
    if (artist == null) {
        artist = readArtistFromDB(name);
        artistCache.put(name, artist);
    }
    return artist;
}
```

Java 8 引入了一个新方法 computeIfAbsent，该方法接受一个 Lambda 表达式，值不存在时使用该 Lambda 表达式计算新值。使用该方法，可将上述代码重写为例 5-32 所示的形式。

例 5-32　使用 computeIfAbsent 缓存

```
public Artist getArtist(String name) {
    return artistCache.computeIfAbsent(name, this::readArtistFromDB);
}
```

你可能还希望在值不存在时不计算，为 Map 接口新增的 compute 和 computeIfAbsent 就能处理这些情况。

在工作中，你可能尝试过在 Map 上迭代。过去的做法是使用 value 方法返回一个值的集合，然后在集合上迭代。这样的代码不易读。例 5-33 展示了本章早些时候介绍的一种方式，创建一个 Map，然后统计每个艺术家专辑的数量。

例 5-33　一种丑陋的迭代 Map 的方式

```
Map<Artist, Integer> countOfAlbums = new HashMap<>();
for(Map.Entry<Artist, List<Album>> entry : albumsByArtist.entrySet()) {
    Artist artist = entry.getKey();
    List<Album> albums = entry.getValue();
    countOfAlbums.put(artist, albums.size());
}
```

谢天谢地，Java 8 为 Map 接口新增了一个 forEach 方法，该方法接受一个 BiConsumer 对象为参数（该对象接受两个参数，返回空），通过内部迭代编写出易于阅读的代码，关于内部迭代请参考 3.1 节。使用该方法重写后的代码如例 5-34 所示。

例 5-34　使用内部迭代遍历 Map 里的值

```
Map<Artist, Integer> countOfAlbums = new HashMap<>();
albumsByArtist.forEach((artist, albums) -> {
    countOfAlbums.put(artist, albums.size());
});
```

5.5　要点回顾

- 方法引用是一种引用方法的轻量级语法，形如：ClassName::methodName。
- 收集器可用来计算流的最终值，是 reduce 方法的模拟。
- Java 8 提供了收集多种容器类型的方式，同时允许用户自定义收集器。

5.6　练习

1. 方法引用

回顾第 3 章中的例子，使用方法引用改写以下方法：

a. 转换大写的 map 方法；
b. 使用 reduce 实现 count 方法；
c. 使用 flatMap 连接列表。

2. 收集器

a. 找出名字最长的艺术家，分别使用收集器和第 3 章介绍过的 reduce 高阶函数实现。

然后对比二者的异同：哪一种方式写起来更简单，哪一种方式读起来更简单？以下面的参数为例，该方法的正确返回值为 "Stuart Sutcliffe"：

```
Stream<String> names = Stream.of("John Lennon", "Paul McCartney",
     "George Harrison", "Ringo Starr", "Pete Best", "Stuart Sutcliffe");
```

b. 假设一个元素为单词的流，计算每个单词出现的次数。假设输入如下，则返回值为一个形如 [John → 3, Paul → 2, George → 1] 的 Map：

```
Stream<String> names = Stream.of("John", "Paul", "George", "John",
                                 "Paul", "John");
```

c. 用一个定制的收集器实现 Collectors.groupingBy 方法，不需要提供一个下游收集器，只需实现一个最简单的即可。别看 JDK 的源码，这是作弊！提示：可从下面这行代码开始：

```
public class GroupingBy<T, K> implements Collector<T, Map<K, List<T>>, Map<K,
List<T>>>
```

这是一个进阶练习，不妨最后再尝试这道习题。

3. 改进Map

使用 Map 的 computeIfAbsent 方法高效计算斐波那契数列。这里的“高效”是指避免将那些较小的序列重复计算多次。

第 6 章

数据并行化

前面多次提到，在 Java 8 中，编写并行化的程序很容易。这都多亏了第 3 章介绍的 Lambda 表达式和流，我们完全不必理会串行或并行，只要告诉程序该做什么就行了。这听起来和长久以来使用 Java 编程的方式并无区别，但告诉计算机做什么和怎么做是完全不同的。

从外部迭代到内部迭代的过渡（详见第 3 章），确实让编写简洁的代码更加容易，但这还不是唯一的好处，另一个好处是程序员不需要手动控制迭代过程了。迭代过程不是非要串行化，通过改动一个方法调用来告诉计算机我们的意图，就会出现一个类库指明我们怎么做。

代码的改动微不足道，因此本章主要内容并不在于如何更改代码，而是讲述为什么需要并行化和什么时候会带来性能的提升。要提醒大家的是，本章并不是关于 Java 性能的泛泛之谈，我们只关注 Java 8 轻松提升性能的技术。

6.1　并行和并发

快速浏览一下本书的目录结构，读者可能就会发现本章的标题含有*并行*字样，而第 9 章的标题则带有*并发*字样。别担心，我并不是为了多挣点稿费而将同一个主题写了两次。并发和并行是两个不同的概念，它们的作用也不一样。

并发是两个任务共享时间段，并行则是两个任务在同一时间发生，比如运行在多核 CPU 上。如果一个程序要运行两个任务，并且只有一个 CPU 给它们分配了不同的时间片，那么这就是并发，而不是并行。两者之间的区别如图 6-1 所示。

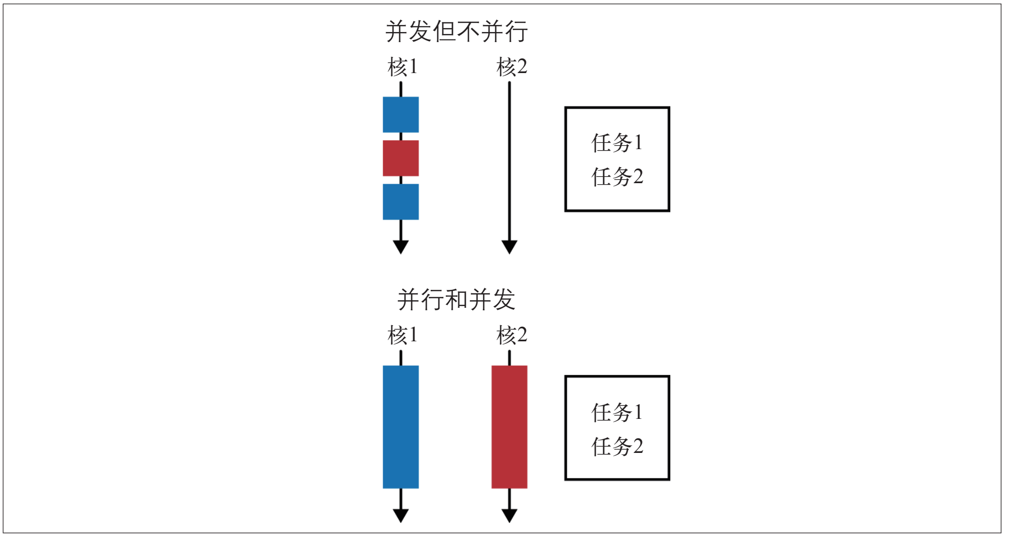


图 6-1：并发和并行的区别

并行化是指为缩短任务执行时间，将一个任务分解成几部分，然后并行执行。这和顺序执行的任务量是一样的，区别就像用更多的马来拉车，花费的时间自然减少了。实际上，和顺序执行相比，并行化执行任务时，CPU 承载的工作量更大。

本章会讨论一种特殊形式的并行化：*数据并行化*。数据并行化是指将数据分成块，为每块数据分配单独的处理单元。还是拿马拉车那个例子打比方，就像从车里取出一些货物，放到另一辆车上，两辆马车都沿着同样的路径到达目的地。

当需要在大量数据上执行同样的操作时，数据并行化很管用。它将问题分解为可在多块数据上求解的形式，然后对每块数据执行运算，最后将各数据块上得到的结果汇总，从而获得最终答案。

人们经常拿*任务并行化*和数据并行化做比较，在任务并行化中，线程不同，工作各异。我们最常遇到的 Java EE 应用容器便是任务并行化的例子之一，每个线程不光可以为不同用户服务，还可以为同一个用户执行不同的任务，比如登录或往购物车添加商品。

6.2　为什么并行化如此重要

过去我们可以指望 CPU 时钟频率会变得越来越快。1979 年，英特尔公司推出的 8086 处理器的时钟频率为 5 MHz；到了 1993 年，奔腾芯片的速度达到了 60 MHz。在 21 世纪早期，CPU 的处理速度一直以这种方式增长。

然而在过去十年中，主流的芯片厂商转向了多核处理器。在写作本书时，服务器通过几个

物理单元搭载 32 或 64 核的情况已不鲜见，而且，这种趋势尚无减弱的征兆。

这种变化影响到了软件设计。我们不能再依赖提升 CPU 的时钟频率来提高现有代码的计算能力，需要利用现代 CPU 的架构，而这唯一的办法就是编写并行化的代码。

大家若已经听过这个消息，我该是多么欣慰。事实上，这一观点在过去几年中，不断地被各种会议的演讲者、技术图书的作者和顾问提及。阿姆达尔定律让我开始关注并行化的重要性。

阿姆达尔定律是一个简单规则，预测了搭载多核处理器的机器提升程序速度的理论最大值。以一段完全串行化的程序为例，如果将其一半改为并行化处理，则不管增加多少处理器，其理论上的最大速度只是原来的 2 倍。有了大量的处理器后，现在这已经是现实了，问题的求解时间将完全取决于它可被分解成几个部分。

以这样的方式思考性能问题，优化任何和计算相关的任务立即变成了如何有效利用现有硬件的问题。当然，不是所有的任务都和计算相关，本章只关注这类和计算相关的问题。

6.3 并行化流操作

并行化操作流只需改变一个方法调用。如果已经有一个 `Stream` 对象，调用它的 `parallel` 方法就能让其拥有并行操作的能力。如果想从一个集合类创建一个流，调用 `parallelStream` 就能立即获得一个拥有并行能力的流。

让我们先来看一个具体的例子，例 6-1 计算了一组专辑的曲目总长度。它拿到每张专辑的曲目信息，然后得到曲目长度，最后相加得出曲目总长度。

例 6-1　串行化计算专辑曲目长度

```
public int serialArraySum() {
    return albums.stream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

调用 `parallelStream` 方法即能并行处理，如例 6-2 所示，剩余代码都是一样的，并行化就是这么简单！

例 6-2　并行化计算专辑曲目长度

```
public int parallelArraySum() {
    return albums.parallelStream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

读到这里，大家的第一反应可能是立即将手头代码中的 `stream` 方法替换为 `parallelStream` 方法，因为这样做简直太简单了！先别忙，为了将硬件物尽其用，利用好并行化非常重要，但流类库提供的数据并行化只是其中的一种形式。

我们先要问自己一个问题：并行化运行基于流的代码是否比串行化运行更快？这不是一个简单的问题。回到前面的例子，哪种方式花的时间更多取决于串行或并行化运行时的环境。

以例 6-1 和例 6-2 中的代码为准，在一个四核电脑上，如果有 10 张专辑，串行化代码的速度是并行化代码速度的 8 倍；如果将专辑数量增至 100 张，串行化和并行化速度相当；如果将专辑数量增值 10 000 张，则并行化代码的速度是串行化代码速度的 2.5 倍。

> 本章的对比基准只是为了说明问题，如果读者尝试在自己的机器上重现这些实验，得到的结果可能会跟书中的结果大相径庭。

输入流的大小并不是决定并行化是否会带来速度提升的唯一因素，性能还会受到编写代码的方式和核的数量的影响。6.6 节会详述和性能有关的细节，但现在还是再来看一个更复杂的例子吧。

6.4 模拟系统

并行化流操作的用武之地是使用简单操作处理大量数据，比如模拟系统。本节我们会搭建一个简易的模拟系统来理解摇骰子，但其中的原理对于大型、真实的系统也适用。

我们这里要讨论的是*蒙特卡洛模拟法*。蒙特卡洛模拟法会重复相同的模拟很多次，每次模拟都使用随机生成的种子。每次模拟的结果都被记录下来，汇总得到一个对系统的全面模拟。蒙特卡洛模拟法被大量用在工程、金融和科学计算领域。

如果公平地掷两次骰子，然后将赢的一面上的点数相加，就会得到一个 2~12 的数字。点数的和至少是 2，因为骰子六个面上最小的点数是 1，而我们将骰子掷了两次；点数的和最大超不过 12，因为骰子点数最多的一面也不过 6 点。我们想要得出点数落在 2~12 之间每个值的概率。

解决该问题的方法之一是求出掷骰子的所有组合，比如，得到 2 点的方式是第一次掷得 1 点，第二次也掷得 1 点。总共有 36 种可能的组合，因此，掷得 2 点的概率就是 1/36。

另外一种解法是使用 1 到 6 的随机数模拟掷骰子事件，然后用得到每个点数的次数除以总的投掷次数。这就是一个简单的蒙特卡洛模拟。模拟投掷骰子的次数越多，得到的结果越

准确，因此，我们希望尽可能多地增加模拟次数。

例 6-3 展示了如何使用流实现蒙特卡洛模拟法。N 代表模拟次数，在❶处使用 IntStream 的 range 方法创建大小为 N 的流，在❷处调用 parallel 方法使用流的并行化操作，twoDiceThrows 函数模拟了连续掷两次骰子事件，返回值是两次点数之和。在❸处使用 mapToObj 方法以便在流上使用该函数。

例 6-3　使用蒙特卡洛模拟法并行化模拟掷骰子事件

```
public Map<Integer, Double> parallelDiceRolls() {
    double fraction = 1.0 / N;
    return IntStream.range(0, N)                        ❶
                    .parallel()                         ❷
                    .mapToObj(twoDiceThrows())          ❸
                    .collect(groupingBy(side -> side,   ❹
                        summingDouble(n -> fraction))); ❺
}
```

在❹处得到了需要合并的所有结果的流，使用前一章介绍的 groupingBy 方法将点数一样的结果合并。我说过要计算每个点数的出现次数，然后除以总的模拟次数 N。在流框架中，将数字映射为 1/N 并且相加很简单，这和前面说的计算方法是等价的。在❺处我们使用 summingDouble 方法完成了这一步。最终的返回值类型是 Map<Integer, Double>，是点数之和到它们的概率的映射。

我得承认这段代码不算儿戏，但使用 5 行代码即能实现蒙特卡洛模拟法还是很精巧的。重要的是模拟的次数越多，得到的结果越准确，因此我们运行多次模拟的动机就会更加强烈。这是一个很好的并行化案列，并行化能带来速度的提升。

我已经带领读者浏览了整个实现细节，为了对比，例 6-4 给出了手动实现并行化蒙特卡洛模拟法的代码。可以看到，大多数代码都在处理调度和等待线程池中的某项任务完成。而使用并行化的流时，这些都不用程序员手动管理。

例 6-4　通过手动使用线程模拟掷骰子事件

```
public class ManualDiceRolls {

    private static final int N = 100000000;

    private final double fraction;
    private final Map<Integer, Double> results;
    private final int numberOfThreads;
    private final ExecutorService executor;
    private final int workPerThread;

    public static void main(String[] args) {
        ManualDiceRolls roles = new ManualDiceRolls();
        roles.simulateDiceRoles();
    }
```

```
public ManualDiceRolls() {
    fraction = 1.0 / N;
    results = new ConcurrentHashMap<>();
    numberOfThreads = Runtime.getRuntime().availableProcessors();
    executor = Executors.newFixedThreadPool(numberOfThreads);
    workPerThread = N / numberOfThreads;
}

public void simulateDiceRoles() {
    List<Future<?>> futures = submitJobs();
    awaitCompletion(futures);
    printResults();
}

private void printResults() {
    results.entrySet()
           .forEach(System.out::println);
}

private List<Future<?>> submitJobs() {
    List<Future<?>> futures = new ArrayList<>();
    for (int i = 0; i < numberOfThreads; i++) {
        futures.add(executor.submit(makeJob()));
    }
    return futures;
}

private Runnable makeJob() {
    return () -> {
        ThreadLocalRandom random = ThreadLocalRandom.current();
        for (int i = 0; i < workPerThread; i++) {
            int entry = twoDiceThrows(random);
            accumulateResult(entry);
        }
    };
}

private void accumulateResult(int entry) {
    results.compute(entry, (key, previous) ->
        previous == null ? fraction
                         : previous + fraction
    );
}

private int twoDiceThrows(ThreadLocalRandom random) {
    int firstThrow = random.nextInt(1, 7);
    int secondThrow = random.nextInt(1, 7);
    return firstThrow + secondThrow;
}

private void awaitCompletion(List<Future<?>> futures) {
    futures.forEach((future) -> {
        try {
            future.get();
        } catch (InterruptedException | ExecutionException e) {
```

```
                e.printStackTrace();
            }
        });
        executor.shutdown();
    }

}
```

6.5 限制

之前提到过使用并行流能工作，但这样说有点无耻。虽然只需一点改动，就能让已有代码并行化运行，但前提是代码写得符合约定。为了发挥并行流框架的优势，写代码时必须遵守一些规则和限制。

之前调用 `reduce` 方法，初始值可以为任意值，为了让其在并行化时能工作正常，初值必须为组合函数的*恒等值*。拿恒等值和其他值做 `reduce` 操作时，其他值保持不变。比如，使用 `reduce` 操作求和，组合函数为 `(acc, element) -> acc + element`，则其初值必须为 0，因为任何数字加 0，值不变。

`reduce` 操作的另一个限制是组合操作必须符合*结合律*。这意味着只要序列的值不变，组合操作的顺序不重要。有点疑惑？别担心！请看例 6-5，我们可以改变加法和乘法的顺序，但结果是一样的。

例 6-5　加法和乘法满足结合律

```
(4 + 2) + 1 = 4 + (2 + 1) = 7
(4 * 2) * 1 = 4 * (2 * 1) = 8
```

要避免的是持有锁。流框架会在需要时，自己处理同步操作，因此程序员没有必要为自己的数据结构加锁。如果你执意为流中要使用的数据结构加锁，比如操作的原始集合，那么有可能是自找麻烦。

在前面我还解释过，使用 `parallel` 方法能轻易将流转换为并行流。如果读者在阅读本书的同时，还查看了相应的 API，那么可能会发现还有一个叫 `sequential` 的方法。在要对流求值时，不能同时处于两种模式，要么是并行的，要么是串行的。如果同时调用了 `parallel` 和 `sequential` 方法，最后调用的那个方法起效。

6.6 性能

在前面我简要提及了影响并行流是否比串行流快的一些因素，现在让我们仔细看看它们。理解哪些能工作、哪些不能工作，能帮助在如何使用、什么时候使用并行流这一问题上做出明智的决策。影响并行流性能的主要因素有 5 个，依次分析如下。

- 数据大小

  输入数据的大小会影响并行化处理对性能的提升。将问题分解之后并行化处理，再将结果合并会带来额外的开销。因此只有数据足够大、每个数据处理管道花费的时间足够多时，并行化处理才有意义。6.3 节讨论过。

- 源数据结构

  每个管道的操作都基于一些初始数据源，通常是集合。将不同的数据源分割相对容易，这里的开销影响了在管道中并行处理数据时到底能带来多少性能上的提升。

- 装箱

  处理基本类型比处理装箱类型要快。

- 核的数量

  极端情况下，只有一个核，因此完全没必要并行化。显然，拥有的核越多，获得潜在性能提升的幅度就越大。在实践中，核的数量不单指你的机器上有多少核，更是指运行时你的机器能使用多少核。这也就是说同时运行的其他进程，或者线程关联性（强制线程在某些核或 CPU 上运行）会影响性能。

- 单元处理开销

  比如数据大小，这是一场并行执行花费时间和分解合并操作开销之间的战争。花在流中每个元素身上的时间越长，并行操作带来的性能提升越明显。

使用并行流框架，理解如何分解和合并问题是很有帮助的。这让我们能够知悉底层如何工作，但却不必了解框架的细节。

来看一个具体的问题，看看如何分解和合并它。例 6-6 是并行求和的代码。

例 6-6　并行求和

```
private int addIntegers(List<Integer> values) {
    return values.parallelStream()
                 .mapToInt(i -> i)
                 .sum();
}
```

在底层，并行流还是沿用了 fork/join 框架。fork 递归式地分解问题，然后每段并行执行，最终由 join 合并结果，返回最后的值。

图 6-2 形象地展示了例 6-6 中代码所示的操作。

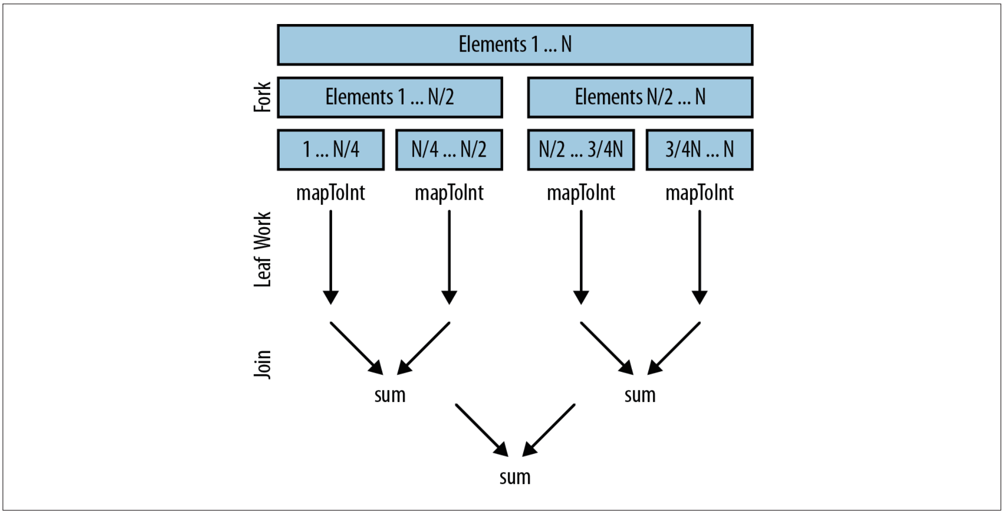


图 6-2：使用 fork/join 分解合并问题

假设并行流将我们的工作分解开，在一个四核的机器上并行执行。

1. 数据被分成四块。
2. 如 6-6 所示，计算工作在每个线程里并行执行。这包括将每个 `Integer` 对象映射为 `int` 值，然后在每个线程里将 1/4 的数字相加。理想情况下，我们希望在这里花的时间越多越好，因为这里是并行操作的最佳场所。
3. 然后合并结果。在例 6-6 中，就是 `sum` 操作，但这也可能是 `reduce`、`collect` 或其他终结操作。

根据问题的分解方式，初始的数据源的特性变得尤其重要，它影响了分解的性能。直观上看，能重复将数据结构对半分解的难易程度，决定了分解操作的快慢。能对半分解同时意味着待分解的值能够被等量地分解。

我们可以根据性能的好坏，将核心类库提供的通用数据结构分成以下 3 组。

- 性能好

  `ArrayList`、数组或 `IntStream.range`，这些数据结构支持随机读取，也就是说它们能轻而易举地被任意分解。

- 性能一般

  `HashSet`、`TreeSet`，这些数据结构不易公平地被分解，但是大多数时候分解是可能的。

- 性能差

  有些数据结构难于分解，比如，可能要花 $O(N)$ 的时间复杂度来分解问题。其中包括 `LinkedList`，对半分解太难了。还有 `Streams.iterate` 和 `BufferedReader.lines`，它们长度未知，因此很难预测该在哪里分解。

初始的数据结构影响巨大。举一个极端的例子，对比对 10 000 个整数并行求和，使用 `ArrayList` 要比使用 `LinkedList` 快 10 倍。这不是说业务逻辑的性能情况也会如此，只是说明了数据结构对于性能的影响之大。使用形如 `LinkedList` 这样难于分解的数据结构并行运行可能更慢。

理想情况下，一旦流框架将问题分解成小块，就可以在每个线程里单独处理每一小块，线程之间不再需要进一步通信。无奈现实不总遂人愿！

在讨论流中单独操作每一块的种类时，可以分成两种不同的操作：*无状态的*和*有状态的*。无状态操作整个过程中不必维护状态，有状态操作则有维护状态所需的开销和限制。

如果能避开有状态，选用无状态操作，就能获得更好的并行性能。无状态操作包括 `map`、`filter` 和 `flatMap`，有状态操作包括 `sorted`、`distinct` 和 `limit`。

> 要对自己的代码进行性能测试。本节只给出了哪些性能特征需要调查，但什么都比不上实际的测试和分析。

6.7 并行化数组操作

Java 8 还引入了一些针对数组的并行操作，脱离流框架也可以使用 Lambda 表达式。像流框架上的操作一样，这些操作也都是针对数据的并行化操作。让我们看看如何使用这些操作解决那些使用流框架难以解决的问题。

这些操作都在工具类 `Arrays` 中，该类还包括 Java 以前版本中提供的和数组相关的有用方法，表 6-1 总结了新增的并行化操作。

表6-1：数组上的并行化操作

方法名	操　　作
`parallelPrefix`	任意给定一个函数，计算数组的和
`parallelSetAll`	使用 Lambda 表达式更新数组元素
`parallelSort`	并行化对数组元素排序

读者可能以前写过类似例 6-7 的代码，使用一个 `for` 循环初始化数组。在这里，我们用数组下标初始化数组中的每个元素。

例 6-7　使用 `for` 循环初始化数组

```
public static double[] imperativeInitilize(int size) {
    double[] values = new double[size];
    for(int i = 0; i < values.length;i++) {
        values[i] = i;
    }
    return values;
}
```

使用 parallelSetAll 方法能轻松地并行化该过程，代码如例 6-8 所示。首先提供了一个用于操作的数组，然后传入一个 Lambda 表达式，根据数组下标计算元素的值。在该例中，数组下标和元素的值是一样的。使用这些方法有一点要小心：它们改变了传入的数组，而没有创建一个新的数组。

例 6-8　使用并行化数组操作初始化数组

```
public static double[] parallelInitialize(int size) {
    double[] values = new double[size];
    Arrays.parallelSetAll(values, i -> i);
    return values;
}
```

parallelPrefix 操作擅长对时间序列数据做累加，它会更新一个数组，将每一个元素替换为当前元素和其前驱元素的和，这里的“和”是一个宽泛的概念，它不必是加法，可以是任意一个 BinaryOperator。

使用该方法能计算的例子之一是一个简单的滑动平均数。在时间序列上增加一个滑动窗口，计算出窗口中的平均值。如果输入数据为 0、1、2、3、4、3.5，滑动窗口的大小为 3，则简单滑动平均数为 1、2、3、3.5。例 6-9 展示了如何计算滑动平均数。

例 6-9　计算简单滑动平均数

```
public static double[] simpleMovingAverage(double[] values, int n) {
    double[] sums = Arrays.copyOf(values, values.length); ❶
    Arrays.parallelPrefix(sums, Double::sum); ❷
    int start = n - 1;
    return IntStream.range(start, sums.length) ❸
                    .mapToDouble(i -> {
                        double prefix = i == start ? 0 : sums[i - n];
                        return (sums[i] - prefix) / n; ❹
                    })
                    .toArray(); ❺
}
```

这段代码有点复杂，我会分步介绍它是如何工作的。参数 n 是时间窗口的大小，我们据此计算滑动平均值。由于要使用的并行操作会改变数组内容，为了不修改原有数据，在❶处复制了一份输入数据。

在❷处执行并行操作，将数组的元素相加。现在 sums 变量中保存了求和结果。比如输入 0、1、2、3、4、3.5，则计算后的值为 0.0、1.0、3.0、6.0、10.0、13.5。

现在有了和，就能计算出时间窗口中的和了，减去窗口起始位置的元素即可，除以 n 即得到平均值。可以使用已有的流中的方法计算该值，那就让我们来试试吧！使用 Intstream.range 得到包含所需元素下标的流。

在❹处使用总和减去窗口起始值，然后再除以 n 得到平均值。最后在❺处将流转换为数组。

6.8 要点回顾

- 数据并行化是把工作拆分，同时在多核 CPU 上执行的方式。
- 如果使用流编写代码，可通过调用 `parallel` 或者 `parallelStream` 方法实现数据并行化操作。
- 影响性能的五要素是：数据大小、源数据结构、值是否装箱、可用的 CPU 核数量，以及处理每个元素所花的时间。

6.9 练习

1. 例 6-10 中的代码顺序求流中元素的平方和，将其改为并行处理。

例 6-10　顺序求列表中数字的平方和

```
public static int sequentialSumOfSquares(IntStream range) {
    return range.map(x -> x * x)
                .sum();
}
```

2. 例 6-11 中的代码把列表中的数字相乘，然后再将所得结果乘以 5。顺序执行这段程序没有问题，但并行执行时有一个缺陷，使用流并行化执行该段代码，并修复缺陷。

例 6-11　把列表中的数字相乘，然后再将所得结果乘以 5，该实现有一个缺陷

```
public static int multiplyThrough(List<Integer> linkedListOfNumbers) {
    return linkedListOfNumbers.stream()
                  .reduce(5, (acc, x) -> x * acc);
}
```

3. 例 6-12 中的代码计算列表中数字的平方和。尝试改进代码性能，但不得牺牲代码质量。只需要一些简单的改动即可。

例 6-12　求列表元素的平方和，该实现方式性能不高

```
public int slowSumOfSquares() {
    return linkedListOfNumbers.parallelStream()
                              .map(x -> x * x)
                              .reduce(0, (acc, x) -> acc + x);
}
```

确保将基准代码运行多次，GitHub 上提供的示例代码有一份基准数据可供使用。

第 7 章 测试、调试和重构

重构、测试驱动开发（TDD）和持续集成（CI）越来越流行，如果我们需要将 Lambda 表达式应用于日常编程工作中，就得学会如何为它编写单元测试。

关于如何测试和调试计算机程序的书已经汗牛充栋，本章不打算再一一赘述。如果读者对如何正确地使用测试驱动开发（TDD）感兴趣，我极力推荐大家阅读 Kent Beck 写的 *Test-Driven Development*，以及由 Steve Freeman 和 Nat Pryce 写的 *Growing Object-Oriented Software, Guided by Tests*（两本书均由 Addison-Wesley 出版社出版）。

本章主要探讨如何在代码中使用 Lambda 表达式的技术，也会说明什么情况下不应该（直接）使用 Lambda 表达式。本章还讲述了如何调试大量使用 Lambda 表达式和流的程序。

先看几个例子，看看如何将现有代码重构为使用 Lambda 表达式的代码。这部分内容前面已经有所涉及，比如在局部范围内的一些重构，使用流操作替代 `for` 循环。本章要讨论的内容更加深入，看看如何使用 Lambda 表达式提高非集合类代码的质量。

7.1 重构候选项

使用 Lambda 表达式重构代码有个时髦的称呼：Lambda 化（读作 lambda-fi-cation，执行重构的程序员叫作 lamb-di-fiers 或者有责任心的程序员）。Java 8 中的核心类库就曾经历过这样一场重构。在选择内部设计模型时，想想以何种形式向外展示 API 是大有裨益的。

这里有一些要点，可以帮助读者确定什么时候应该 Lambda 化自己的应用或类库。其中的每一条都可看作一个局部的反模式或代码异味，借助于 Lambda 化可以修复。

7.1.1　进进出出、摇摇晃晃

例 7-1 是关于如何在程序中记录日志的，我在第 4 章多次提到这个代码。这段代码先调用 `isDebugEnabled` 方法抽取布尔值，用来检查是否启用调试级别，如果启用，则调用 `Logger` 对象的相应方法记录日志。如果你发现自己的代码不断地查询和操作某对象，目的只为了在最后给该对象设个值，那么这段代码就本该属于你所操作的对象。

例 7-1　`logger` 对象使用 `isDebugEnabled` 属性避免不必要的性能开销

```
Logger logger = new Logger();
if (logger.isDebugEnabled()) {
    logger.debug("Look at this: " + expensiveOperation());
}
```

记录日志本来就是一直以来很难实现的目标，因为地方不同，所需的行为也不一样。本例中，需要根据程序中记录日志的不同位置和要记录的内容生成不同的信息。

这种反模式通过传入代码即数据的方式很容易解决。与其查询并设置一个对象的值，不如传入一个 Lambda 表达式，该表达式按照计算得出的值执行相应的行为。我将原来的实现代码列在例 7-2 中，以示提醒。当程序处于调试级别，并且检查是否使用 Lambda 表达式的逻辑被封装在 `Logger` 对象中时，才会调用 Lambda 表达式。

例 7-2　使用 Lambda 表达式简化记录日志代码

```
Logger logger = new Logger();
logger.debug(() -> "Look at this: " + expensiveOperation());
```

上述记录日志的例子也展示了如何使用 Lambda 表达式更好地面向对象编程（OOP），面向对象编程的核心之一是封装局部状态，比如日志的级别。通常这点做得不是很好，`isDebugEnabled` 方法暴露了内部状态。如果使用 Lambda 表达式，外面的代码根本不需要检查日志级别。

7.1.2　孤独的覆盖

这个代码异味是使用继承，其目的只是为了覆盖一个方法。`ThreadLocal` 就是一个很好的例子。`ThreadLocal` 能创建一个工厂，为每个线程最多只产生一个值。这是确保非线程安全的类在并发环境下安全使用的一种简单方式。假设要在数据库中查询一个艺术家，但希望每个线程只做一次这种查询，写出的代码可能如例 7-3 所示。

例 7-3　在数据库中查找艺术家

```
ThreadLocal<Album> thisAlbum = new ThreadLocal<Album> () {
    @Override protected Album initialValue() {
        return database.lookupCurrentAlbum();
    }
};
```

在 Java 8 中，可以为工厂方法 withInitial 传入一个 Supplier 对象的实例来创建对象，如例 7-4 所示。

例 7-4　使用工厂方法

```
ThreadLocal<Album> thisAlbum
    = ThreadLocal.withInitial(() -> database.lookupCurrentAlbum());
```

我们认为第二个例子优于前一个有以下几个原因。首先，任何已有的 Supplier<Album> 实例不需要重新封装，就可以在此使用，这鼓励了重用和组合。

在其他都一样的情况下，代码短小精悍就是个优势。更重要的是，这是代码更加清晰的结果，阅读代码时，信噪比降低了。这意味着有更多时间来解决实际问题，而不是把时间花在继承的样板代码上。这样做还有一个优点，JVM 会少加载一个类。

对每个试图阅读代码，弄明白代码意图的人来说，也清楚了很多。如果你试着大声念出第二个例子中的单词，能很容易听出是干嘛的，但第一个例子就不行了。

有趣的是，在 Java 8 以前，这并不是一个反模式，而是惯用的代码编写方式，就像使用匿名内部类传递行为一样，都不是反模式，而是在 Java 中表达你所想的唯一方式。随着语言的演进，编程习惯也要与时俱进。

7.1.3　同样的东西写两遍

不要重复你劳动（Don't Repeat Yourself，DRY）是一个众所周知的模式，它的反面是*同样的东西写两遍*（Write Everything Twice，WET）。这种代码异味多见于重复的样板代码，产生了更多需要测试的代码，这样的代码难于重构，一改就坏。

不是所有 WET 的情况都适合 Lambda 化。有时，重复是唯一可以避免系统过紧耦合的方式。什么时候该将 WET 的代码 Lambda 化？这里有一个信号可以参考。如果有一个整体上大概相似的模式，只是行为上有所不同，就可以试着加入一个 Lambda 表达式。

让我们看一个更具体的例子。回到我们有关音乐的问题，我想增加一个简单的 Order 类来计算用户购买专辑的一些有用属性，如计算音乐家人数、曲目和专辑时长等。如果使用命令式 Java，编写出的代码如例 7-5 所示。

例 7-5　Order 类的命令式实现

```
public long countRunningTime() {
    long count = 0;
    for (Album album : albums) {
        for (Track track : album.getTrackList()) {
            count += track.getLength();
        }
    }
    return count;
```

```
    }

    public long countMusicians() {
        long count = 0;
        for (Album album : albums) {
            count += album.getMusicianList().size();
        }
        return count;
    }

    public long countTracks() {
        long count = 0;
        for (Album album : albums) {
            count += album.getTrackList().size();
        }
        return count;
    }
```

每个方法里，都有样板代码将每个专辑里的属性和总数相加，比如每首曲目的长度或音乐家的人数。我们没有重用共有的概念，写出了更多代码需要测试和维护。可以使用 `Stream` 来抽象，使用 Java 8 中的集合类库来重写上述代码，使之更紧凑。如果直接将上述命令式的代码翻译成使用流的形式，则形如例 7-6。

例 7-6　使用流重构命令式的 `Order` 类

```
    public long countRunningTime() {
        return albums.stream()
                .mapToLong(album -> album.getTracks()
                                         .mapToLong(track -> track.getLength())
                                         .sum())
                .sum();
    }

    public long countMusicians() {
        return albums.stream()
                .mapToLong(album -> album.getMusicians().count())
                .sum();
    }

    public long countTracks() {
        return albums.stream()
                .mapToLong(album -> album.getTracks().count())
                .sum();
    }
```

然而这段代码仍然有重用可读性的问题，因为有一些抽象和共性只能使用领域内的知识来表达。流不会提供一个方法统计每张专辑上的信息——这是程序员要自己编写的领域知识。这也是在 Java 8 出现之前很难编写的领域方法，因为每个方法都不一样。

想一下如何实现这样一个函数。我们返回一个 `long`，统计所有专辑的某些特征，还需要一个 Lambda 表达式，告诉我们统计专辑上的什么信息。也就是说我们的方法需要一个

参数，该参数为每张专辑返回一个 `long`，方便的是，Java 8 核心类库中已经有了这样一个类型 `ToLongFunction`。如图 7-1 所示，它的类型随参数类型，因此我们要使用的类型为 `ToLongFunction<Album>`。

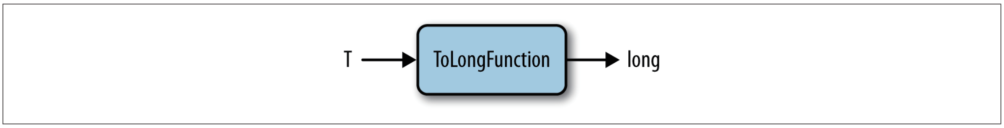


图 7-1：`ToLongFunction`

这些都定下来之后，方法体就自然定下来了。我们将专辑转换成流，将专辑映射为 `long`，然后求和。在实现直接面对客户的代码时，比如 `countTracks`，传入一个代表了领域知识的 Lambda 表达式，在这里，就是将专辑映射为上面的曲目。例 7-7 是使用了这种方式转换之后的代码。

例 7-7　使用领域方法重构 `Order` 类

```
public long countFeature(ToLongFunction<Album> function) {
    return albums.stream()
            .mapToLong(function)
            .sum();
}

public long countTracks() {
    return countFeature(album -> album.getTracks().count());
}

public long countRunningTime() {
    return countFeature(album -> album.getTracks()
                                      .mapToLong(track -> track.getLength())
                                      .sum());
}

public long countMusicians() {
    return countFeature(album -> album.getMusicians().count());
}
```

7.2　Lambda表达式的单元测试

> 单元测试是测试一段代码的行为是否符合预期的方式。

通常，在编写单元测试时，怎么在应用中调用该方法，就怎么在测试中调用。给定一些输入或测试替身，调用这些方法，然后验证结果是否和预期的行为一致。

Lambda 表达式给单元测试带来了一些麻烦，Lambda 表达式没有名字，无法直接在测试代码中调用。

你可以在测试代码中复制 Lambda 表达式来测试，但这种方式的副作用是测试的不是真正的实现。假设你修改了实现代码，测试仍然通过，而实现可能早已在做另一件事了。

解决该问题有两种方式。第一种是将 Lambda 表达式放入一个方法测试，这种方式要测那个方法，而不是 Lambda 表达式本身。例 7-8 是一个将一组字符串转换成大写的方法。

例 7-8　将字符串转换为大写形式

```
public static List<String> allToUpperCase(List<String> words) {
    return words.stream()
                .map(string -> string.toUpperCase())
                .collect(Collectors.<String>toList());
}
```

在这段代码中，Lambda 表达式唯一的作用就是调用一个 Java 方法。将该 Lambda 表达式单独测试是不值得的，它的行为太简单了。

如果换我来测试这段代码，我会将重点放在方法的行为上。比如例 7-9 测试了流中有多个单词的情况，它们都被转换成对应的大写。

例 7-9　测试大写转换

```
@Test
public void multipleWordsToUppercase() {
    List<String> input = Arrays.asList("a", "b", "hello");
    List<String> result = Testing.allToUpperCase(input);
    assertEquals(asList("A", "B", "HELLO"), result);
}
```

有时候 Lambda 表达式实现了复杂的功能，它可能包含多个边界情况、使用了多个属性来计算一个非常重要的值。你非常想测试该段代码的行为，但它是一个 Lambda 表达式，无法引用。

作为例子，让我们来看一个比大写转换更复杂一点的方法。我们要把字符串的第一个字母转换成大写，其他部分保持不变。使用流和 Lambda 表达式，编写的代码形如例 7-10 所示。在❶处使用 Lambda 表达式做转换。

例 7-10　将列表中元素的第一个字母转换成大写

```
public static List<String> elementFirstToUpperCaseLambdas(List<String> words) {
    return words.stream()
            .map(value -> { ❶
                char firstChar = Character.toUpperCase(value.charAt(0));
                return firstChar + value.substring(1);
            })
            .collect(Collectors.<String>toList());
}
```

如果要测试这段代码，我们必须创建一个列表，然后将想要测试的各种情况都测试到。例 7-11 展示了这种方式有多么繁琐，别担心，我们有办法！

例 7-11　测试字符串包含两个字符的情况，第一个字母被转换为大写

```
@Test
public void twoLetterStringConvertedToUppercaseLambdas() {
    List<String> input = Arrays.asList("ab");
    List<String> result = Testing.elementFirstToUpperCaseLambdas(input);
    assertEquals(asList("Ab"), result);
}
```

别用 Lambda 表达式。我知道，在一本介绍如何使用 Lambda 表达式的书里，这个建议有点奇怪，但是方楔子钉不进圆孔。既然如此，大家一定会问如何测试代码，同时享有 Lambda 表达式带来的便利？

*请用*方法引用。任何 Lambda 表达式都能被改写为普通方法，然后使用方法引用直接引用。

例 7-12 将 Lambda 表达式重构为一个方法，然后在主程序中使用，主程序负责转换字符串。

例 7-12　将首字母转换为大写，应用到所有列表元素

```
public static List<String> elementFirstToUppercase(List<String> words) {
    return words.stream()
                .map(Testing::firstToUppercase)
                .collect(Collectors.<String>toList());
}

public static String firstToUppercase(String value) { ❶
    char firstChar = Character.toUpperCase(value.charAt(0));
    return firstChar + value.substring(1);
}
```

把处理字符串的的逻辑抽取成一个方法后，就可以测试该方法，把所有的边界情况都覆盖到。新的测试用例如例 7-13 所示。

例 7-13　测试单独的方法

```
@Test
public void twoLetterStringConvertedToUppercase() {
    String input = "ab";
    String result = Testing.firstToUppercase(input);
    assertEquals("Ab", result);
}
```

7.3　在测试替身时使用Lambda表达式

编写单元测试的常用方式之一是使用*测试替身*描述系统中其他模块的期望行为。这种方式很有用，因为单元测试可以脱离其他模块来测试你的类或方法，测试替身让你能用单元测

试来实现这种隔离。

> 测试替身也常被称为模拟，事实上测试存根和模拟都属于测试替身。区别是模拟可以验证代码的行为。读者若想了解更多有关这方面的信息，请阅读 Martin Fowler 的相关文章（http://martinfowler.com/articles/mocksArentStubs.html）。

测试代码时，使用 Lambda 表达式的最简单方式是实现轻量级的测试存根。如果交互的类本身就是一个函数接口，实现这样的存根就非常简单和自然。

在 7.1.3 节中，讨论过如何将通用的领域逻辑重构为一个 `countFeature` 方法，然后使用 Lambda 表达式实现不同的统计行为。例 7-14 展示了如何对此编写单元测试。

例 7-14　使用 Lambda 表达式编写测试替身，传给 `countFeature` 方法

```
@Test
public void canCountFeatures() {
    OrderDomain order = new OrderDomain(asList(
            newAlbum("Exile on Main St."),
            newAlbum("Beggars Banquet"),
            newAlbum("Aftermath"),
            newAlbum("Let it Bleed")));

    assertEquals(8, order.countFeature(album -> 2));
}
```

对于 `countFeature` 方法的期望行为是为传入的专辑返回某个数值。这里传入 4 张专辑，测试存根中为每张专辑返回 2，然后断言该方法返回 8，即 2×4。如果要向代码传入一个 Lambda 表达式，最好确保 Lambda 表达式也通过测试。

多数的测试替身都很复杂，使用 *Mockito* 这样的框架有助于更容易地产生测试替身。让我们考虑一种简单情形，为 `List` 生成测试替身。我们不想返回 `List` 本上的长度，而是返回另一个 `List` 的长度，为了模拟 `List` 的 `size` 方法，我们不想只给出答案，还想做一些操作，因此传入一个 Lambda 表达式，如例 7-15 所示。

例 7-15　结合 Mockito 框架使用 Lambda 表达式

```
List<String> list = mock(List.class);

when(list.size()).thenAnswer(inv -> otherList.size());

assertEquals(3, list.size());
```

Mockito 使用 `Answer` 接口允许用户提供其他行为，换句话说，这是我们的老朋友：代码即数据。之所以在这里能使用 Lambda 表达式，是因为 `Answer` 本身就是一个函数接口。

7.4　惰性求值和调试

调试时通常会设置断点，单步跟踪程序的每一步。使用流时，调试可能会变得更加复杂，因为迭代已交由类库控制，而且很多流操作是惰性求值的。

在传统的命令式编程看来，代码就是达到某种目的的一系列行动，在行动前后查看程序状态是有意义的。在 Java 8 中，你仍然可以使用 IDE 提供的各种调试工具，但有时需要调整实现方式，以期达到更好的结果。

7.5　日志和打印消息

假设你要在集合上进行大量操作，你要调试代码，你希望看到每一步操作的结果是什么。可以在每一步打印出集合中的值，这在流中很难做到，因为一些中间步骤是惰性求值的。

让我们通过第 3 章介绍的命令式版本的国际报告程序，看看如何记录中间值。考虑到读者可能已经忘记这个程序，我们再来解释一下这个程序的意图，该程序找出了专辑上每位艺术家来自哪个国家。在例 7-16 中，我们将找到的国家信息记录到日志中。

例 7-16　记录中间值，以便调试 `for` 循环

```
Set<String> nationalities = new HashSet<>();
for (Artist artist : album.getMusicianList()) {
    if (artist.getName().startsWith("The")) {
        String nationality = artist.getNationality();
        System.out.println("Found nationality: " + nationality);
        nationalities.add(nationality);
    }
}
return nationalities;
```

现在可以使用 `forEach` 方法打印出流中的值，这同时会触发求值过程。但是这样的操作有个缺点：我们无法再继续操作流了，流只能使用一次。如果我们还想继续，必须重新创建流。例 7-17 展示了这样的代码会有多难看。

例 7-17　使用 `forEach` 记录中间值，这种方式有点幼稚

```
album.getMusicians()
     .filter(artist -> artist.getName().startsWith("The"))
     .map(artist -> artist.getNationality())
     .forEach(nationality -> System.out.println("Found: " + nationality));

Set<String> nationalities
    = album.getMusicians()
           .filter(artist -> artist.getName().startsWith("The"))
           .map(artist -> artist.getNationality())
           .collect(Collectors.<String>toSet());
```

7.6 解决方案：peak

遗憾的是，流有一个方法让你能查看每个值，同时能继续操作流。这就是 peek 方法。例 7-18 使用 peek 方法重写了前面的例子，输出流中的值，同时避免了重复的流操作。

例 7-18　使用 peek 方法记录中间值

```
Set<String> nationalities
    = album.getMusicians()
           .filter(artist -> artist.getName().startsWith("The"))
           .map(artist -> artist.getNationality())
           .peek(nation -> System.out.println("Found nationality: " + nation))
           .collect(Collectors.<String>toSet());
```

使用 peek 方法还能以同样的方式，将输出定向到现有的日志系统中，比如 log4j、java.util.logging 或者 slf4j。

7.7 在流中间设置断点

记录日志这是 peek 方法的用途之一。为了像调试循环那样一步一步跟踪，可在 peek 方法中加入断点，这样就能逐个调试流中的元素了。

此时，peek 方法可知包含一个空的方法体，只要能设置断点就行。有一些调试器不允许在空的方法体中设置断点，此时，我将值简单地映射为其本身，这样就有地方设置断点了，虽然这样做不够完美，但只要能工作就行。

7.8 要点回顾

- 重构遗留代码时考虑如何使用 Lambda 表达式，有一些通用的模式。
- 如果想要对复杂一点的 Lambda 表达式编写单元测试，将其抽取成一个常规的方法。
- peek 方法能记录中间值，在调试时非常有用。

第 8 章

设计和架构的原则

软件开发最重要的设计工具不是什么技术，而是一颗在设计原则方面训练有素的头脑。

—— Craig Larman

通过前面的学习，我们认识到 Lambda 表达式是对 Java 语言的一点简单改进，在 JDK 标准类库中，运行它的方式各种各样。但是大多数 Java 代码都不是由开发 JDK 的程序员写的，而是像你我这样的普通程序员。为了最大限度发挥 Lambda 表达式的优势，大家需要将其引入已有代码中。作为一名职业 Java 程序员，Lambda 表达式没有什么特别的，和接口、类一样，它只是程序员工具箱中的一件新工具。

本章将探索如何使用 Lambda 表达式实现 SOLID 原则，该原则是开发良好面向对象程序的准则。使用 Lambda 表达式，还能改进一些现有的设计模式，本章也会为大家简要介绍几个这样的例子。

和同事一起工作时，肯定会遇到这样的情况：你实现了一个新功能或修复了一个缺陷，并且对自己的修改很满意。但其他人看了你的代码后——也许发生在代码审查环节，完全不买账！对于什么是好代码，什么是坏代码，存在分歧很正常。

大多数时候，人们意见不统一，是他们各自都有自己的想法。审查你代码的人可能会选择另一种实现方式，这并不是说你们俩谁对谁错。引入 Lambda 表达式后，又多了一个话题。这并不是说该功能本身有多复杂，或者需要花大力气去争论，而是人们在讨论设计问题时又多了一项谈资。

本章旨在帮助大家写出优秀的程序，我会给出一些良好的设计原则和模式，在此基础之

上，就能开发出可维护且十分可靠的程序。我们不光会用到 JDK 提供的崭新类库，而且会教大家如何在自己的领域和应用程序中使用 Lambda 表达式。

8.1 Lambda表达式改变了设计模式

*设计模式*是人们熟悉的另一种设计思想，它是软件架构中解决通用问题的模板。如果碰到一个问题，并且恰好熟悉一个与之适应的模式，就能直接应用该模式来解决问题。从某种程度上来说，设计模式将解决特定问题的最佳实践途径固定了下来。

当然，没有永远的最佳实践。以曾经风靡一时的单例模式为例，该模式确保只产生一个对象实例。在过去十年中，人们批评它让程序变得更脆弱，且难于测试。敏捷开发的流行，让测试显得更加重要，单例模式的这个问题把它变成了一个*反模式*：一种应该避免使用的模式。

本书的重点并不是讨论设计模式如何变得过时，相反，我们讨论的是如何使用 Lambda 表达式，让现有设计模式变得更好、更简单，或者在某些情况下，有了不同的实现方式。Java 8 引入的新语言特性是所有这些设计模式变化的推动因素。

8.1.1 命令者模式

*命令者*是一个对象，它封装了调用另一个方法的所有细节，*命令者模式*使用该对象，可以编写出根据运行期条件，顺序调用方法的一般化代码。命令者模式中有四个类参与其中，如图 8-1 所示。

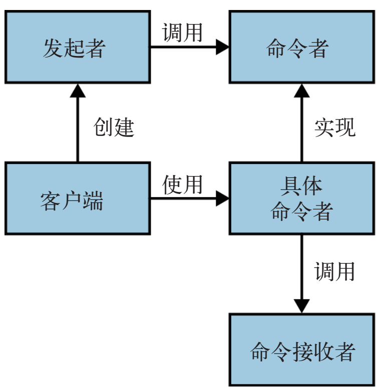


图 8-1：命令者模式

- *命令接收者*
  执行实际任务。

- 命令者
  封装了所有调用命令执行者的信息。

- 发起者
  控制一个或多个命令的顺序和执行。

- 客户端
  创建具体的命令者实例。

看一个命令者模式的具体例子，看看如何使用 Lambda 表达式改进该模式。假设有一个 GUI `Editor` 组件，在上面可以执行 `open`、`save` 等一系列操作，如例 8-1 所示。现在我们想实现宏功能——也就是说，可以将一系列操作录制下来，日后作为一个操作执行，这就是我们的命令接收者。

例 8-1　文本编辑器可能具有的一般功能

```
public interface Editor {

    public void save();

    public void open();

    public void close();

}
```

在该例子中，像 `open`、`save` 这样的操作称为命令，我们需要一个统一的接口来概括这些不同的操作，我将这个接口叫作 `Action`，它代表了一个操作。所有的命令都要实现该接口（例 8-2）。

例 8-2　所有操作均实现 `Action` 接口

```
public interface Action {

    public void perform();

}
```

现在让每个操作都实现该接口，这些类要做的只是在 `Action` 接口中调用 `Editor` 类中的一个方法。我将遵循恰当的命名规范，用类名代表操作，比如 `save` 方法对应 `Save` 类。例 8-3 和例 8-4 是定义好的命令对象。

例 8-3　保存操作代理给 `Editor` 方法

```
public class Save implements Action {

    private final Editor editor;

    public Save(Editor editor) {
```

```
        this.editor = editor;
    }

    @Override
    public void perform() {
        editor.save();
    }
}
```

例 8-4　打开文件操作代理给 Editor 方法

```
public class Open implements Action {

    private final Editor editor;

    public Open(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void perform() {
        editor.open();
    }
}
```

现在可以实现 Macro 类了，该类 record 操作，然后一起运行。我们使用 List 保存操作序列，然后调用 forEach 方法按顺序执行每一个 Action，例 8-5 就是我们的命令发起者。

例 8-5　包含操作序列的宏，可按顺序执行操作

```
public class Macro {

    private final List<Action> actions;

    public Macro() {
        actions = new ArrayList<>();
    }

    public void record(Action action) {
        actions.add(action);
    }

    public void run() {
        actions.forEach(Action::perform);
    }
}
```

在构建宏时，将每一个命令实例加入 Macro 对象的列表，然后运行宏，就会按顺序执行每一条命令。我是个“懒惰的”程序员，喜欢将通用的工作流定义成宏。我说“懒惰”了吗？我的意思其实是提高工作效率。例 8-6 展示了如何在用户代码中使用 Macro 对象。

例 8-6　使用命令者模式构建宏

```
Macro macro = new Macro();
```

```
macro.record(new Open(editor));
macro.record(new Save(editor));
macro.record(new Close(editor));
macro.run();
```

Lambda 表达式能做点什么呢？事实上，所有的命令类，Save、Open 都是 Lambda 表达式，只是暂时藏在类的外壳下。它们是一些行为，我们通过创建类将它们在对象之间传递。Lambda 表达式能让这个模式变得非常简单，我们可以扔掉这些类。例 8-7 展示了去掉命令类，使用 Lambda 表达式后的程序。

例 8-7　使用 Lambda 表达式构建宏

```
Macro macro = new Macro();
macro.record(() -> editor.open());
macro.record(() -> editor.save());
macro.record(() -> editor.close());
macro.run();
```

事实上 ，如果意识到这些 Lambda 表达式的作用只是调用了一个方法，还能让问题变得更简单。我们可以使用方法引用将命令和宏对象关联起来（如例 8-8 所示）。

例 8-8　使用方法引用构建宏

```
Macro macro = new Macro();
macro.record(editor::open);
macro.record(editor::save);
macro.record(editor::close);
macro.run();
```

命令者模式只是一个可怜的程序员使用 Lambda 表达式的起点。使用 Lambda 表达式或是方法引用，能让代码更简洁，去除了大量样板代码，让代码意图更加明显。

宏只是使用命令者模式的一个例子，它被大量用在实现组件化的图形界面系统、撤销功能、线程池、事务和向导中。

> 在核心 Java 中，已经有一个和 Action 接口结构一致的函数接口——Runnable。我们可以在实现上述宏程序中直接使用该接口，但在这个例子中，似乎 Action 是一个更符合我们待解问题的词汇，因此我们创建了自己的接口。

8.1.2　策略模式

策略模式能在运行时改变软件的算法行为。如何实现策略模式根据你的情况而定，但其主要思想是定义一个通用的问题，使用不同的算法来实现，然后将这些算法都封装在一个统一接口的背后。

文件压缩就是一个很好的例子。我们提供给用户各种压缩文件的方式，可以使用 zip 算法，

也可以使用 gzip 算法，我们实现一个通用的 `Compressor` 类，能以任何一种算法压缩文件。

首先，为我们的策略定义 API（参见图 8-2），我把它叫作 `CompressionStrategy`，每一种文件压缩算法都要实现该接口。该接口有一个 `compress` 方法，接受并返回一个 `OutputStream` 对象，返回的就是压缩后的 `OutputStream`（如例 8-9 所示）。

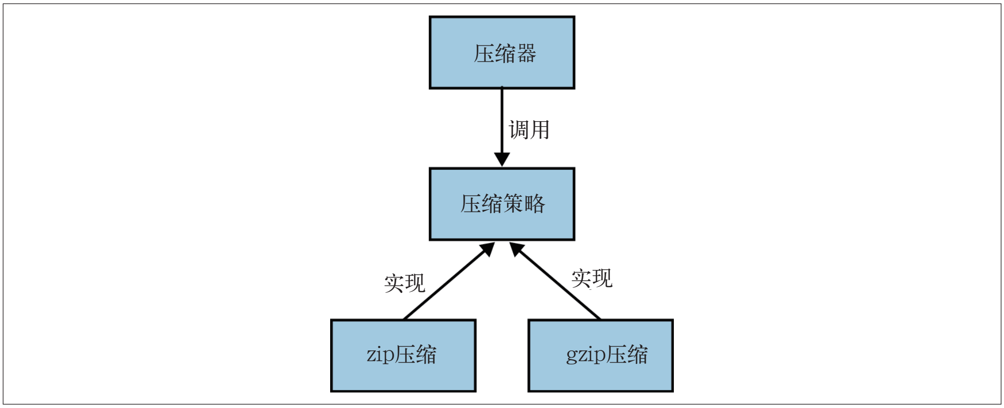


图 8-2：策略模式

例 8-9　定义压缩数据的策略接口

```
public interface CompressionStrategy {

    public OutputStream compress(OutputStream data) throws IOException;

}
```

我们有两个类实现了该接口，分别代表 gzip 和 ZIP 算法，使用 Java 内置的类实现 gzip（例 8-10）和 ZIP（例 8-11）算法。

例 8-10　使用 gzip 算法压缩数据

```
public class GzipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new GZIPOutputStream(data);
    }
}
```

例 8-11　使用 zip 算法压缩数据

```
public class ZipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new ZipOutputStream(data);
    }
}
```

现在可以动手实现 Compressor 类了，这里就是使用策略模式的地方。该类有一个 compress 方法，读入文件，压缩后输出。它的构造函数有一个 CompressionStrategy 参数，调用代码可以在运行期使用该参数决定使用哪种压缩策略，比如，可以等待用户输入选择（如例 8-12 所示）。

例 8-12　在构造类时提供压缩策略

```
public class Compressor {

    private final CompressionStrategy strategy;

    public Compressor(CompressionStrategy strategy) {
        this.strategy = strategy;
    }

    public void compress(Path inFile, File outFile) throws IOException {
        try (OutputStream outStream = new FileOutputStream(outFile)) {
            Files.copy(inFile, strategy.compress(outStream));
        }
    }
}
```

如果使用这种传统的策略模式实现方式，可以编写客户代码创建一个新的 Compressor，并且使用任何我们想要的策略（如例 8-13 所示）。

例 8-13　使用具体的策略类初始化 Compressor

```
Compressor gzipCompressor = new Compressor(new GzipCompressionStrategy());
gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(new ZipCompressionStrategy());
zipCompressor.compress(inFile, outFile);
```

和前面讨论的命令者模式一样，使用 Lambda 表达式或者方法引用可以去掉样板代码。在这里，我们可以去掉具体的策略实现，使用一个方法实现算法，这里的算法由构造函数中对应的 OutputStream 实现。使用这种方式，可以完全舍弃 GzipCompressionStrategy 和 ZipCompressionStrategy 类。例 8-14 展示了使用方法引用后的代码。

例 8-14　使用方法引用初始化 Compressor

```
Compressor gzipCompressor = new Compressor(GZIPOutputStream::new);
gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(ZipOutputStream::new);
zipCompressor.compress(inFile, outFile);
```

8.1.3　观察者模式

观察者模式是另一种可被 Lambda 表达式简化和改进的行为模式。在观察者模式中，被观察者持有一个观察者列表。当被观察者的状态发生改变，会通知观察者。观察者模式被大

量应用于基于 MVC 的 GUI 工具中，以此让模型状态发生变化时，自动刷新视图模块，达到二者之间的解耦。

观看 GUI 模块自动刷新有点枯燥，我们要观察的对象是月球！ NASA 和外星人都对登陆到月球上的东西感兴趣，都希望可以记录这些信息。NASA 希望确保阿波罗号上的航天员成功登月；外星人则希望在 NASA 注意力分散之时进犯地球。

让我们先来定义观察者的 API，这里我将观察者称作 `LandingObserver`。它只有一个 `observeLanding` 方法，当有东西登陆到月球上时会调用该方法（例 8-15）。

例 8-15　用于观察登陆到月球的组织的接口

```
public interface LandingObserver {

    public void observeLanding(String name);

}
```

被观察者是月球 `Moon`，它持有一组 `LandingObserver` 实例，有东西着陆时会通知这些观察者，还可以增加新的 `LandingObserver` 实例观测 `Moon` 对象（例 8-16）。

例 8-16　`Moon` 类——当然不如现实世界中那么完美

```
public class Moon {

    private final List<LandingObserver> observers = new ArrayList<>();

    public void land(String name) {
        for (LandingObserver observer : observers) {
            observer.observeLanding(name);
        }
    }

    public void startSpying(LandingObserver observer) {
        observers.add(observer);
    }
}
```

我们有两个具体的类实现了 `LandingObserver` 接口，分别代表外星人（例 8-17）和 NASA（例 8-18）检测着陆情况。前面提到过，监测到登陆后它们有不同的反应。

例 8-17　外星人观察到人类登陆月球

```
public class Aliens implements LandingObserver {

    @Override
    public void observeLanding(String name) {
        if (name.contains("Apollo")) {
            System.out.println("They're distracted, lets invade earth!");
        }
    }
}
```

例 8-18　NASA 也能观察到有人登陆月球

```
public class Nasa implements LandingObserver {
    @Override
    public void observeLanding(String name) {
        if (name.contains("Apollo")) {
            System.out.println("We made it!");
        }
    }
}
```

和前面的模式类似，在传统的例子中，用户代码需要有一层模板类，如果使用 Lambda 表达式，就不用编写这些类了（如例 8-19 和例 8-20 所示）。

例 8-19　使用类的方式构建用户代码

```
Moon moon = new Moon();
moon.startSpying(new Nasa());
moon.startSpying(new Aliens());

moon.land("An asteroid");
moon.land("Apollo 11");
```

例 8-20　使用 Lambda 表达式构建用户代码

```
Moon moon = new Moon();

moon.startSpying(name -> {
    if (name.contains("Apollo"))
        System.out.println("We made it!");
});

moon.startSpying(name -> {
    if (name.contains("Apollo"))
        System.out.println("They're distracted, lets invade earth!");
});

moon.land("An asteroid");
moon.land("Apollo 11");
```

还有一点值得思考，无论使用观察者模式或策略模式，实现时采用 Lambda 表达式还是传统的类，取决于策略和观察者代码的复杂度。我这里所举的例子代码很简单，只是一两个方法调用，很适合展示新的语言特性。然而在有些情况下，观察者本身就是一个很复杂的类，这时将很多代码塞进一个方法中会大大降低代码的可读性。

> 从某种角度来说，将大量代码塞进一个方法会让可读性变差是决定如何使用 Lambda 表达式的黄金法则。之所以不在这里过分强调，是因为这也是编写一般方法时的黄金法则！

8.1.4 模板方法模式

开发软件时一个常见的情况是有一个通用的算法，只是步骤上略有不同。我们希望不同的实现能够遵守通用模式，保证它们使用了同一个算法，也是为了让代码更加易读。一旦你从整体上理解了算法，就能更容易理解其各种实现。

模板方法模式是为这些情况设计的：整体算法的设计是一个抽象类，它有一系列抽象方法，代表算法中可被定制的步骤，同时这个类中包含了一些通用代码。算法的每一个变种由具体的类实现，它们重写了抽象方法，提供了相应的实现。

让我们假想一个情境来搞明白这是怎么回事。假设我们是一家银行，需要对公众、公司和职员放贷。放贷程序大体一致——验明身份、信用记录和收入记录。这些信息来源不一，衡量标准也不一样。你可以查看一个家庭的账单来核对个人身份；公司都在官方机构注册过，比如美国的 SEC、英国的 Companies House。

我们先使用一个抽象类 `LoanApplication` 来控制算法结构，该类包含一些贷款调查结果报告的通用代码。根据不同的申请人，有不同的类：`CompanyLoanApplication`、`PersonalLoanApplication` 和 `EmployeeLoanApplication`。例 8-21 展示了 `LoanApplication` 类的结构。

例 8-21　使用模板方法模式描述申请贷款过程

```java
public abstract class LoanApplication {

    public void checkLoanApplication() throws ApplicationDenied {
        checkIdentity();
        checkCreditHistory();
        checkIncomeHistory();
        reportFindings();
    }

    protected abstract void checkIdentity() throws ApplicationDenied;

    protected abstract void checkIncomeHistory() throws ApplicationDenied;

    protected abstract void checkCreditHistory() throws ApplicationDenied;

    private void reportFindings() {
```

`CompanyLoanApplication` 的 `checkIdentity` 方法在 Companies House 等注册公司数据库中查找相关信息。`checkIncomeHistory` 方法评估公司的现有利润、损益表和资产负债表。`checkCreditHistory` 方法则查看现有的坏账和未偿债务。

`PersonalLoanApplication` 的 `checkIdentity` 方法通过分析客户提供的纸本结算单，确认客户地址是否真实有效。`checkIncomeHistory` 方法通过检查工资条判断客户是否仍被雇佣。`checkCreditHistory` 方法则会将工作交给外部的信用卡支付提供商。

`EmployeeLoanApplication` 就是没有查阅员工历史功能的 `PersonalLoanApplication`。为了方

便起见，我们的银行在雇佣员工时会查阅所有员工的收入记录（例 8-22）。

例 8-22　员工申请贷款是个人申请的一种特殊情况

```
public class EmployeeLoanApplication extends PersonalLoanApplication {

    @Override
    protected void checkIncomeHistory() {
        // 这是自己人！
    }
}
```

使用 Lambda 表达式和方法引用，我们能换个角度思考模板方法模式，实现方式也跟以前不一样。模板方法模式真正要做的是将一组方法调用按一定顺序组织起来。如果用函数接口表示函数，用 Lambda 表达式或者方法引用实现这些接口，相比使用继承构建算法，就会得到极大的灵活性。让我们看看如何使用这种方式实现 LoanApplication 算法，请看例 8-23！

例 8-23　员工申请贷款的例子

```
public class LoanApplication {

    private final Criteria identity;
    private final Criteria creditHistory;
    private final Criteria incomeHistory;

    public LoanApplication(Criteria identity,
                           Criteria creditHistory,
                           Criteria incomeHistory) {

        this.identity = identity;
        this.creditHistory = creditHistory;
        this.incomeHistory = incomeHistory;
    }

    public void checkLoanApplication() throws ApplicationDenied {
        identity.check();
        creditHistory.check();
        incomeHistory.check();
        reportFindings();
    }

    private void reportFindings() {
```

正如读者所见，这里没有使用一系列的抽象方法，而是多出一些属性：identity、creditHistory 和 incomeHistory。每一个属性都实现了函数接口 Criteria，该接口检查一项标准，如果不达标就抛出一个问题域里的异常。我们也可以选择从 check 方法返回一个类来表示成功或失败，但是沿用异常更加符合先前的实现（如例 8-24 所示）。

例 8-24　如果申请失败，函数接口 Criteria 抛出异常

```
public interface Criteria {
```

```
        public void check() throws ApplicationDenied;

    }
```

采用这种方式，而不是基于继承的模式的好处是不需要在 LoanApplication 及其子类中实现算法，分配功能时有了更大的灵活性。比如，我们想让 Company 类负责所有的检查，那么 Company 类就会多出一系列方法，如例 8-25 所示。

例 8-25 Company 类中的检查方法

```
public void checkIdentity() throws ApplicationDenied;

public void checkProfitAndLoss() throws ApplicationDenied;

public void checkHistoricalDebt() throws ApplicationDenied;
```

现在只需为 CompanyLoanApplication 类传入对应的方法引用，如例 8-26 所示。

例 8-26 CompanyLoanApplication 类声明了对应的检查方法

```
public class CompanyLoanApplication extends LoanApplication {

    public CompanyLoanApplication(Company company) {
        super(company::checkIdentity,
              company::checkHistoricalDebt,
              company::checkProfitAndLoss);
    }
}
```

将行为分配给 Company 类的原因是各个国家之间确认公司信息的方式不同。在英国，Companies House 规范了注册公司信息的地址，但在美国，各个州的政策是不一样的。

使用函数接口实现检查方法并没有排除继承的方式。我们可以显式地在这些类中使用 Lambda 表达式或者方法引用。

我们也不需要强制 EmployeeLoanApplication 继承 PersonalLoanApplication 来达到复用，可以对同一个方法传递引用。它们之间是否天然存在继承关系取决于员工的借贷是否是普通人借贷这种特殊情况，或者是另外一种不同类型的借贷。因此，使用这种方式能让我们更加紧密地为问题建模。

8.2 使用Lambda表达式的领域专用语言

领域专用语言（DSL）是针对软件系统中某特定部分的编程语言。它们通常比较小巧，表达能力也不如 Java 这样能应对大多数编程任务的通用语言强。DSL 高度专用：不求面面俱到，但求有所专长。

人们通常将 DSL 分为两类：内部 DSL 和外部 DSL。外部 DSL 脱离程序源码编写，然后单独解析和实现。比如级联样式表（CSS）和正则表达式，就是常用的外部 DSL。

内部 DSL 嵌入编写它们的编程语言中。如果读者使用过 JMock 和 Mockito 等模拟类库，或用过 SQL 构建 API，如 JOOQ 或 Querydsl，那么就知道什么是内部 DSL。从某种角度上说，内部 DSL 就是普通的类库，提供 API 方便使用。虽然简单，内部 DSL 却功能强大，让你的代码变得更加精炼、易读。理想情况下，使用 DSL 编写的代码读起来就像描述问题所使用的语言。

有了 Lambda 表达式，实现 DSL 就更简单了，那些想尝试 DSL 的程序员又多了一件趁手的工具。我们将通过实现一个用于行为驱动开发（BDD）的 DSL：LambdaBehave，来探索其中遇到的各种问题。

BDD 是测试驱动开发（TDD）的一个变种，它的重点是描述程序的行为，而非一组需要通过的单元测试。我们的设计灵感源于一个叫 Jasmine 的 JavaScript BDD 框架，前端开发中会大量使用该框架。例 8-27 展示了如何使用 Jasmine 创建测试用例。

例 8-27　Jasmine

```
describe("A suite is just a function", function() {
    it("and so is a spec", function() {
        var a = true;

        expect(a).toBe(true);
    });
});
```

如果读者不熟悉 JavaScript，阅读这段代码可能会稍感疑惑。下面我们使用 Java 8 实现一个类似的框架时会一步一步来，只需要记住，在 JavaScript 中我们使用 `function() { … }` 来表示 Lambda 表达式。

让我们分别来看看这些概念：

- 每一个规则描述了程序的一种行为；
- 期望是描述应用行为的一种方式，在规则中定义；
- 多个规则合在一起，形成一个套件。

这些概念在传统的测试框架，比如 JUnit 中，都有对应的概念。规则对应一个测试方法，期望对应断言，套件对应一个测试类。

8.2.1　使用Java编写DSL

让我们先看一下实现后的 Java BDD 框架长什么样子，例 8-28 描述了一个 `Stack` 的某些行为。

例 8-28　描述 `Stack` 的案例

```
public class StackSpec {{

    describe("a stack", it -> {

        it.should("be empty when created", expect -> {
            expect.that(new Stack()).isEmpty();
```

```
        });

        it.should("push new elements onto the top of the stack", expect -> {
            Stack<Integer> stack = new Stack<>();
            stack.push(1);

            expect.that(stack.get(0)).isEqualTo(1);
        });

        it.should("pop the last element pushed onto the stack", expect -> {
            Stack<Integer> stack = new Stack<>();
            stack.push(2);
            stack.push(1);

            expect.that(stack.pop()).isEqualTo(2);
        });

    });

}}
```

首先我们使用动词 describe 为套件起头，然后定义一个名字表明这是描述什么东西的行为，这里我们使用了 "a stack"。

每一条规则读起来尽可能接近英语中的句子。它们均以 it.should 打头，其中 it 指正在描述的对象。然后用一句简单的英语描述行为，最后使用 expect.that 做前缀，描述期待的行为。

检查规则时，会从命令行得到一个简单的报告，表明是否有规则失败。你会发现 pop 操作期望的返回值是 2，而不是 1，因此“pop the last element pushed onto the stack”这条规则就失败了：

```
a stack
    should pop the last element pushed onto the stack[expected:❶ but was:❷  ]
    should be empty when created
    should push new elements onto the top of the stack
```

8.2.2 实现

读者已经领略了使用 Lambda 表达式的 DSL 所带来的便利，现在该看看我们是如何实现该框架的。我们希望会让大家看到，自己实现一个这样的框架是多么简单。

描述行为首先看到的是 describe 这个动词，简单导入一个静态方法就够了。为套件创建一个 Description 实例，在此处理各种各样的规则。Description 类就是我们定义的 DSL 中的 it（详见例 8-29）。

例 8-29　从 describe 方法开始定义规则

```
public static void describe(String name, Suite behavior) {
    Description description = new Description(name);
    behavior.specifySuite(description);
}
```

每个套件的规则描述由用户使用一个 Lambda 表达式实现，因此我们需要一个 Suite 函数接口来表示规则组成的套件，如例 8-30 所示。该接口接收一个 Description 对象作为参数，我们在 describe 方法里将其传入。

例 8-30　每个测试套件都由一个实现该接口的 Lambda 表达式实现

```
public interface Suite {

    public void specifySuite(Description description);

}
```

在我们定义的 DSL 中，不仅套件由 Lambda 表达式实现，每一条规则也是一个 Lambda 表达式。它们也需要定义一个函数接口：Specification（如例 8-31 所示）。示例代码中的 expect 变量是 Expect 类的实例，我们稍后描述：

例 8-31　每条规则都是一个实现该接口的 Lambda 表达式

```
public interface Specification {

    public void specifyBehaviour(Expect expect);

}
```

之前来回传递的 Description 实例这里就派上用场了。我们希望用户可以使用 it.should 命名他们的规则，这就是说 Description 类需要有一个 should 方法（如例 8-32 所示）。这里是真正做事的地方，该方法通过调用 specifySuite 执行 Lambda 表达式。如果规则失败，会抛出一个标准的 Java AssertionError，而其他任何 Throwable 对象则认为是一个错误：

例 8-32　将用 Lambda 表达式表示的规则传入 should 方法

```
public void should(String description, Specification specification) {
    try {
        Expect expect = new Expect();
        specification.specifyBehaviour(expect);
        Runner.current.recordSuccess(suite, description);
    } catch (AssertionError cause) {
        Runner.current.recordFailure(suite, description, cause);
    } catch (Throwable cause) {
        Runner.current.recordError(suite, description, cause);
    }
}
```

规则通过 expect.that 描述期望的行为，也就是说 Expect 类需要一个 that 方法供用户调用，如例 8-33 所示。这里可以封装传入的对象，然后暴露一些常用的方法，如 isEqualTo。如果规则失败，抛出相应的断言。

例 8-33　期望链的开始

```
public final class Expect {
```

```
    public BoundExpectation that(Object value) {
        return new BoundExpectation(value);
    }

    // 省去类定义的其他部分
```

读者可能会注意到，我一直忽略了一个细节，该细节与 Lambda 表达式无关。StackSpec 类并没有直接实现任何方法，我直接将代码写在里边。这里我偷了个懒，在类定义的开头和结尾使用了双括号：

```
public class StackSpec {{
    ...
}}
```

这其实是一个匿名构造函数，可以执行任意的 Java 代码块，所以这等价于一个完整的构造函数，只是少了一些样板代码。这段代码也可以写作：

```
public class StackSpec {
    public StackSpec() {
        ...
    }
}
```

要实现一个完整的 BDD 框架还有很多工作要做，本节只是为了向读者展示如何使用 Lambda 表达式创建领域专用语言。我在这里讲解了与 DSL 中 Lambda 表达式交互的部分，以期能帮助读者管中窥豹，了解如何实现这种类型的 DSL。

8.2.3 评估

流畅性的一方面表现在 DSL 是否是 IDE 友好的。换句话说，你只需记住少量知识，然后用代码自动补全功能补齐代码。这就是使用 Description 和 Expect 对象的原因。当然也可以导入静态方法 it 或 expect，一些 DSL 中就使用了这种方式。如果选择向 Lambda 表达式传入对象，而不是导入一个静态方法，就能让 IDE 的使用者轻松补全代码。

用户唯一要记住的是调用 describe 方法，这种方式的好处通过单纯阅读可能无法体会，我建议大家创建一个示例项目，亲自体验这个框架。

另一个值得注意的是大多数测试框架提供了大量注释，或者很多外部“魔法”，或者借助于反射。我们不需要这些技巧，就能直接使用 Lambda 表达式在 DSL 中表达行为，就和使用普通的 Java 方法一样。

8.3 使用Lambda表达式的SOLID原则

SOLID 原则是设计面向对象程序时的一些基本原则。原则的名字是个简写，分别代表了下面五个词的首字母：Single responsibility、Open/closed、Liskov substitution、Interface

segregation 和 Dependency inversion。这些原则能指导你开发出易于维护和扩展的代码。

每种原则都对应着一系列潜在的代码异味，并为其提供了解决方案。有很多图书介绍这个主题，因此我不会详细讲解，而是关注如何在 Lambda 表达式的环境下应用其中的三条原则。在 Java 8 中，有些原则通过扩展，已经超出了原来的限制。

8.3.1　单一功能原则

程序中的类或方法只能有一个改变的理由。

软件开发中不可避免的情况是需求的改变。这可能是需要增加新功能，也可能是你对问题的理解或者客户发生变化了，或者你想变得更快，总之，软件会随着时间不断演进。

当软件的需求发生变化，实现这些功能的类和方法也需要变化。如果你的类有多个功能，一个功能引发的代码变化会影响该类的其他功能。这可能会引入缺陷，还会影响代码演进的能力。

让我们看一个简单的示例程序，该程序由资产列表生成 `BalanceSheet` 表格，然后输出成一份 PDF 格式的报告。如果实现时将制表和输出功能都放进同一个类，那么该类就有两个变化的理由。你可能想改变输出功能，输出不同的格式，比如 HTML，可能还想改变 `BalanceSheet` 的细节。这为将问题分解成两个类提供了很好的理由：一个负责将 `BalanceSheet` 生成表格，一个负责输出。

单一功能原则不止于此：一个类不仅要功能单一，而且还需将功能封装好。换句话说，如果我想改变输出格式，那么只需改动负责输出的类，而不必关心负责制表的类。

这是强*内聚性*设计的一部分。说一个类是内聚的，是指它的方法和属性需要统一对待，因为它们紧密相关。如果你试着将一个内聚的类拆分，可能会得到刚才创建的那两个类。

既然你已经知道了什么是单一功能原则，问题来了：这和 Lambda 表达式有什么关系？

Lambda 表达式在方法级别能更容易实现单一功能原则。让我们看一个例子，该段程序能得出一定范围内有多少个质数（例 8-34）。

例 8-34　计算质数个数，一个方法里塞进了多重职责

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        boolean isPrime = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                isPrime = false;
            }
        }
        if (isPrime) {
```

```
            tally++;
        }
    }
    return tally;
}
```

很显然，在例 8-34 中我们同时干了两件事：计数和判断一个数是否是质数。在例 8-35 中，通过简单重构，将两个功能一分为二。

例 8-35　将 isPrime 重构成另外一个方法后，计算质数个数的方法

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        if (isPrime(i)) {
            tally++;
        }
    }
    return tally;
}

private boolean isPrime(int number) {
    for (int i = 2; i < number; i++) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}
```

但我们的代码还是有两个功能。代码中的大部分都在对数字循环，如果我们遵守单一功能原则，那么迭代过程应该封装起来。改进代码还有一个现实的原因，如果需要对一个很大的 upTo 计数，我们希望可以并行操作。没错，线程模型也是代码的职责之一！

我们可以使用 Java 8 的集合流（如例 8-36 所示）重构上述代码，将循环操作交给类库本身处理。这里使用了 range 方法从 0 至 upTo 计数，然后 filter 出质数，最后对结果做 count。

例 8-36　使用集合流重构质数计数程序

```
public long countPrimes(int upTo) {
    return IntStream.range(1, upTo)
                    .filter(this::isPrime)
                    .count();
}

private boolean isPrime(int number) {
    return IntStream.range(2, number)
                    .allMatch(x -> (number % x) != 0);
}
```

如果我们想利用更多 CPU 加速计数操作，可使用 parallelStream 方法，而不需要修改任何其他代码（如例 8-37 所示）。

例 8-37　并行运行基于集合流的质数计数程序

```
public long countPrimes(int upTo) {
    return IntStream.range(1, upTo)
                    .parallel()
                    .filter(this::isPrime)
                    .count();
}

private boolean isPrime(int number) {
    return IntStream.range(2, number)
            .allMatch(x -> (number % x) != 0);
}
```

因此，利用高阶函数，可以轻松帮助我们实现功能单一原则。

8.3.2　开闭原则

> 软件应该对扩展开放，对修改闭合。
>
> —— Bertrand Meyer

开闭原则的首要目标和单一功能原则类似：让软件易于修改。一个新增功能或一处改动，会影响整个代码，容易引入新的缺陷。开闭原则保证已有的类在不修改内部实现的基础上可扩展，这样就努力避免了上述问题。

第一次听说开闭原则时，感觉有点痴人说梦。不改变实现怎么能扩展一个类的功能呢？答案是借助于抽象，可插入新的功能。让我们看一个具体的例子。

我们要写的程序用来衡量系统性能，并且把得到的结果绘制成图形。比如，我们有描述计算机花在用户空间、内核空间和输入输出上的时间散点图。我将负责显示这些指标的类叫作 `MetricDataGraph`。

设计 `MetricDataGraph` 类的方法之一是将代理收集到的各项指标放入该类，该类的公开 API 如例 8-38 所示。

例 8-38　`MetricDataGraph` 类的公开 API

```
class MetricDataGraph {

    public void updateUserTime(int value);

    public void updateSystemTime(int value);

    public void updateIoTime(int value);

}
```

但这样的设计意味着每次想往散点图中添加新的时间点，都要修改 `MetricDataGraph` 类。通

过引入抽象可以解决这个问题，我们使用一个新类 TimeSeries 来表示各种时间点。这时，MetricDataGraph 类的公开 API 就得以简化，不必依赖于某项具体指标，如例 8-39 所示。

例 8-39　MetricDataGraph 类简化之后的 API

```
class MetricDataGraph {

    public void addTimeSeries(TimeSeries values);

}
```

每项具体指标现在可以实现 TimeSeries 接口，在需要时能直接插入。比如，我们可能会有如下类：UserTimeSeries、SystemTimeSeries 和 IoTimeSeries。如果要添加新的，比如由于虚拟化所浪费的 CPU 时间，则可增加一个新的实现了 TimeSeries 接口的类：StealTimeSeries。这样，就扩展了 MetricDataGraph 类，但并没有修改它。

高阶函数也展示出了同样的特性：对扩展开放，对修改闭合。前面提到的 ThreadLocal 类就是一个很好的例子。ThreadLocal 有一个特殊的变量，每个线程都有一个该变量的副本并与之交互。该类的静态方法 withInitial 是一个高阶函数，传入一个负责生成初始值的 Lambda 表达式。

这符合开闭原则，因为不用修改 ThreadLocal 类，就能得到新的行为。给 withInitial 方法传入不同的工厂方法，就能得到拥有不同行为的 ThreadLocal 实例。比如，可以使用 ThreadLocal 生成一个 DateFormatter 实例，该实例是线程安全的，如例 8-40 所示。

例 8-40　ThreadLocal 日期格式化器

```
// 实现
ThreadLocal<DateFormat> localFormatter
    = ThreadLocal.withInitial(() -> new SimpleDateFormat());

// 使用
DateFormat formatter = localFormatter.get();
```

通过传入不同的 Lambda 表达式，可以得到完全不同的行为。比如在例 8-41 中，我们为每个 Java 线程创建了唯一、有序的标识符。

例 8-41　ThreadLocal 标识符

```
// 或者这样实现
AtomicInteger threadId = new AtomicInteger();
ThreadLocal<Integer> localId
    = ThreadLocal.withInitial(() -> threadId.getAndIncrement());

// 使用
int idForThisThread = localId.get();
```

对开闭原则的另外一种理解和传统的思维不同，那就是使用不可变对象实现开闭原则。不可变对象是指一经创建就不能改变的对象。

“不可变性”一词有两种解释：*观测不可变性*和*实现不可变性*。观测不可变性是指在其他对象看来，该类是不可变的；实现不可变性是指对象本身不可变。实现不可变性意味着观测不可变性，反之则不一定成立。

`java.lang.String` 宣称是不可变的，但事实上只是观测不可变，因为它在第一次调用 `hashCode` 方法时缓存了生成的散列值。在其他类看来，这是完全安全的，它们看不出散列值是每次在构造函数中计算出来的，还是从缓存中返回的。

之所以在这样一本讲解 Lambda 表达式的书中谈及不可变对象，是因为它们都是函数式编程中耳熟能详的概念，这里也是 Lambda 表达式的发源地。它们生来就符合我在本书中讲述的编程风格。

我们说不可变对象实现了开闭原则，是因为它们的内部状态无法改变，可以安全地为其增加新的方法。新增加的方法无法改变对象的内部状态，因此对修改是闭合的；但它们又增加了新的行为，因此对扩展是开放的。当然，你还需留意不要改变程序其他部分的状态。

因其天生线程安全的特性，不可变对象引起了人们的格外注意。它们没有内部状态可变，因此可以安全地在不同线程之间共享。

如果我们回顾这几种方式，会发现已经偏离了传统的开闭原则。事实上，在 Bertrand Meyer 第一次引入这个原则时，原意是一旦实现后，类就不允许改动了。在现代敏捷开发环境中，完成一个类的说法很明显已经过时了。业务需求和使用方法的变化可能会让一个类的功能和当初设计的不同。当然这不成为忽视这一原则的理由，只是说明了所谓的原则只应作为指导，而不应教条地全盘接受，走向极端。

我认为还有一点值得思考，在 Java 8 中，使用抽象插入多个类，或者使用高阶函数来实现开闭原则其实是一样的。因为抽象需要使用一个接口或抽象类来定义方法，这其实就是一种多态的使用方式。

在 Java 8 中，任何传入高阶函数的 Lambda 表达式都由一个函数接口表示，高阶函数负责调用其唯一的方法，根据传入 Lambda 表达式的不同，行为也不同。这其实也是在用多态来实现开闭原则。

8.3.3 依赖反转原则

> *抽象不应依赖细节，细节应该依赖抽象。*

让程序变得死板、脆弱、难于改变的方法之一是将上层业务逻辑和底层粘合模块的代码混在一起，因为这两样东西都会随着时间发生变化。

依赖反转原则的目的是让程序员脱离底层粘合代码，编写上层业务逻辑代码。这就让上层

代码依赖于底层细节的抽象，从而可以重用上层代码。这种模块化和重用方式是双向的：既可以替换不同的细节重用上层代码，也可以替换不同的业务逻辑重用细节的实现。

让我们看一个具体的、自动化构建地址簿的例子，实现时使用了依赖反转原则达到上层的解耦。该应用以电子卡片作为输入，使用某种存储机制编写地址簿。

显然，我们可将代码分成如下三个基本模块：

- 一个能解析电子卡片格式的电子卡片阅读器；
- 能将地址存为文本文件的地址簿存储模块；
- 从电子卡片中获取有效信息并将其写入地址簿的编写模块。

我们用图 8-3 来表示各模块之间的关系。

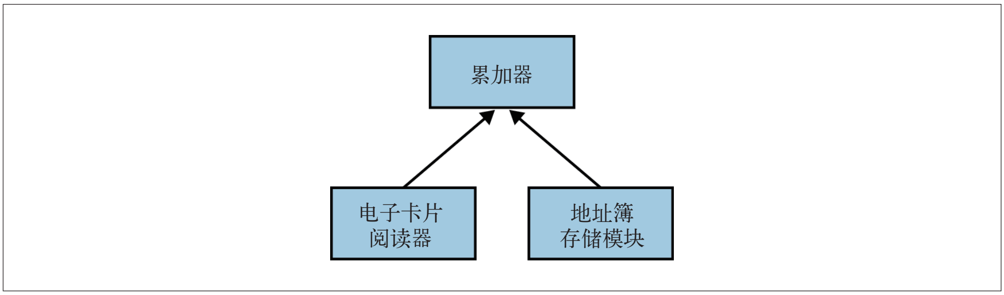


图 8-3：依赖关系

在该系统中，重用编写模块很复杂，但是电子卡片阅读器和地址簿存储模块都不依赖于其他模块，因此很容易在其他系统中重用。还可以替换它们，比如用一个其他的阅读器，或者从人们的 Twitter 账户信息中读取内容；又比如我们不想将地址簿存为一个文本文件，而是使用数据库存储等其他形式。

为了具备能在系统中替换组件的灵活性，必须保证编写模块不依赖阅读器或存储模块的实现细节。因此我们引入了对阅读信息和输出信息的抽象，编写模块的实现依赖于这种抽象。在运行时传入具体的实现细节，这就是依赖反转原则的工作原理。

具体到 Lambda 表达式，我们之前遇到的很多高阶函数都符合依赖反转原则。比如 `map` 函数重用了在两个集合之间转换的代码。`map` 函数不依赖于转换的细节，而是依赖于抽象的概念。在这里，就是依赖函数接口：`Function`。

资源管理是依赖反转的另一个更为复杂的例子。显然，可管理的资源很多，比如数据库连接、线程池、文件和网络连接。这里我将以文件为例，因为文件是一种相对简单的资源，但是背后的原则可以很容易应用到更复杂的资源中。

让我们看一段代码，该段代码从一种假想的标记语言中提取标题，其中标题以冒号（：）

结尾。我们的方法先读取文件，逐行检查，滤出标题，然后关闭文件。我们还将和读写文件有关的异常封装成接近待解决问题的异常：HeadingLookupException，最后的代码如例 8-42 所示。

例 8-42　解析文件中的标题

```
public List<String> findHeadings(Reader input) {
    try (BufferedReader reader = new BufferedReader(input)) {
        return reader.lines()
                     .filter(line -> line.endsWith(":"))
                     .map(line -> line.substring(0, line.length() - 1))
                     .collect(toList());
    } catch (IOException e) {
        throw new HeadingLookupException(e);
    }
}
```

可惜，我们的代码将提取标题和资源管理、文件处理混在一起。我们真正想要的是编写提取标题的代码，而将操作文件相关的细节交给另一个方法。可以使用 Stream<String> 作为抽象，让代码依赖它，而不是文件。Stream 对象更安全，而且不容易被滥用。我们还想传入一个函数，在读文件出问题时，可以创建一个问题域里的异常。整个过程如例 8-43 所示，而且我们将问题域里的异常处理和资源管理的异常处理分开了。

例 8-43　剥离了文件处理功能后的业务逻辑

```
public List<String> findHeadings(Reader input) {
    return withLinesOf(input,
                       lines -> lines.filter(line -> line.endsWith(":"))
                                     .map(line -> line.substring(0, line.length()-1))
                                     .collect(toList()),
                       HeadingLookupException::new);
}
```

是不是想知道 withLinesOf 方法是什么样的？请看例 8-44。

例 8-44　定义 withLinesOf 方法

```
private <T> T withLinesOf(Reader input,
                          Function<Stream<String>, T> handler,
                          Function<IOException, RuntimeException> error) {

    try (BufferedReader reader = new BufferedReader(input)) {
        return handler.apply(reader.lines());
    } catch (IOException e) {
        throw error.apply(e);
    }
}
```

withLinesOf 方法接受一个 Reader 参数处理文件读写，然后将其封装进一个 BufferedReader 对象，这样就可以逐行读取文件了。handler 函数代表了我们想在该方法中执行的代码，它以文件中的每一行组成的 Stream 作为参数。另一个参数是 error，输入输出有异

常时会调用该方法，它会构建出与问题域有关的异常，出问题时就抛出该异常。

总结下来，高阶函数提供了反转控制，这就是依赖反转的一种形式，可以很容易地和 Lambda 表达式一起使用。依赖反转原则另外值得注意的一点是待依赖的抽象不必是接口。这里我们使用 `Stream` 对原始的 `Reader` 和文件处理做抽象，这种方式也适用于函数式编程语言中的资源管理——通常使用高阶函数管理资源，接受一个回调函数使用打开的资源，然后再关闭资源。事实上，如果 Java 7 就有 Lambda 表达式，那么 Java 7 中的 try-with-resources 功能可能只需要一个库函数就能实现。

8.4 进阶阅读

本章讨论的很多内容都涉及了更广泛的设计问题，关注程序整体，而不是一个方法。限于本书讨论的重点是 Lambda 表达式，我们对这些话题的讨论都是浅尝辄止。如果读者想了解更多细节，可参考相关图书。

长期以来，"Bob 大叔"是 SOLID 原则的推动者，他撰写了大量有关该主题的文章和书籍，也多次就该主题举行过演讲。如果你想免费从他那里获取一些相关知识，可访问 Object Mentor 官方网站（http://www.objectmentor.com/resources/publishedArticles.html），在"设计模式"主题下有一系列详述设计原则的文章。

如果你想深入理解领域专用语言，包括内部领域专用语言和外部领域专用语言，推荐大家阅读 Martin Fowler 和 Rebecca Parsons 合著的 *Domain-Specific Languages*（Addison-Wesley 出版社出版）一书。

8.5 要点回顾

- Lambda 表达式能让很多现有设计模式更简单、可读性更强，尤其是命令者模式。
- 在 Java 8 中，创建领域专用语言有更多的灵活性。
- 在 Java 8 中，有应用 SOLID 原则的新机会。

第 9 章

使用Lambda表达式编写并发程序

前面讨论了如何并行化处理数据，本章讨论如何使用 Lambda 表达式编写并发应用，高效传递信息和非阻塞式 I/O。

本章的一些例子用到了 Vert.x（http://vertx.io/）和 RxJava（https://github.com/Netflix/RxJava）框架，但其中展现的设计原则是通用的，对其他框架或是自己编写的、没有使用任何框架的程序也适用。

9.1　为什么要使用非阻塞式I/O

在介绍并行化处理时，讲了很多关于如何高效利用多核 CPU 的内容。这种方式很管用，但在处理大量数据时，它并不是唯一可用的线程模型。

假设要编写一个支持大量用户的聊天程序。每当用户连接到聊天服务器时，都要和服务器建立一个 TCP 连接。使用传统的线程模型，每次向用户写数据时，都要调用一个方法向用户传输数据，这个方法会阻塞当前线程。

这种 I/O 方式叫*阻塞式* I/O，是一种通用且易于理解的方式，因为和程序用户的交互通常符合这样一种顺序执行的方式。缺点是，将系统扩展至支持大量用户时，需要和服务器建立大量 TCP 连接，因此扩展性不是很好。

非阻塞式 I/O，有时也叫*异步* I/O，可以处理大量并发网络连接，而且一个线程可以为多个连接服务。和阻塞式 I/O 不同，对聊天程序客户端的读写调用立即返回，真正的读写操作则在另一个独立的线程执行，这样就可以同时执行其他任务了。如何使用这些省下来的

CPU 周期完全取决于程序员，可以选择读入更多数据，也可以玩一局 Minecraft 游戏。

到目前为止，我避免使用代码来描述这两种 I/O 方式，因为根据 API 的不同，它们有多种实现方式。Java 标准类库的 NIO 提供了非阻塞式 I/O 的接口，NIO 的最初版本用到了 `Selector` 的概念，让一个线程管理多个通信管道，比如向客户端写数据的网络套接字。

然而这种方式压根儿就没有在 Java 程序员中流行起来，它编写出来的代码难于理解和调试。引入 Lambda 表达式后，设计和实现没有这些缺点的 API 就顺手多了。

9.2 回调

为了展示非阻塞式 I/O 的原则，我们将运行一个极其简单的聊天应用，没有那些花里胡哨的功能。当用户第一次连接应用时，需要设定用户名，随后便可通过应用收发信息。

我们将使用 Vert.x 框架实现该应用，并且在实施过程中根据需要，引入其他一些必需的技术。让我们先来写一段接收 TCP 连接的代码，如例 9-1 所示。

例 9-1 接收 TCP 连接

```
public class ChatVerticle extends Verticle {

    public void start() {
        vertx.createNetServer()
             .connectHandler(socket -> {
                 container.logger().info("socket connected");
                 socket.dataHandler(new User(socket, this));
              }).listen(10_000);

        container.logger().info("ChatVerticle started");
    }

}
```

读者可将 `Verticle` 想成 `Servlet`——它是 Vert.x 框架中部署的原子单元。上述代码的入口是 `start` 方法，它和普通 Java 程序中的 `main` 方法类似。在聊天应用中，我们用它建立一个接收 TCP 连接的服务器。

然后向 `connectHandler` 方法输入一个 Lambda 表达式，每当有用户连接到聊天应用时，都会调用该 Lambda 表达式。这就是一个回调，与在第 1 章中介绍的 Swing 中的回调类似。这种方式的好处是，应用不必控制线程模型——Vert.x 框架为我们管理线程，打理好了一切相关复杂性，程序员只需考虑事件和回调就够了。

我们的应用还通过 `dataHandler` 方法注册了另外一个回调，每当从网络套接字读取数据时，该回调就会被调用。在本例中，我们希望提供更复杂的功能，因此没有使用 Lambda 表达式，而是传入一个常规的 `User` 类，该类实现了相关的函数接口。`User` 类的定义如例 9-2 所示。

例 9-2　处理用户连接

```
public class User implements Handler<Buffer> {

    private static final Pattern newline = Pattern.compile("\\n");

    private final NetSocket socket;
    private final Set<String> names;
    private final EventBus eventBus;

    private Optional<String> name;

    public User(NetSocket socket, Verticle verticle) {
        Vertx vertx = verticle.getVertx();

        this.socket = socket;
        names = vertx.sharedData().getSet("names");
        eventBus = vertx.eventBus();
        name = Optional.empty();
    }

    @Override
    public void handle(Buffer buffer) {
        newline.splitAsStream(buffer.toString())
               .forEach(line -> {
                   if (!name.isPresent())
                       setName(line);
                   else
                       handleMessage(line);
                });
    }

    // Class continues...
```

变量 buffer 包含了网络连接写入的数据，我们使用的是一个分行的文本协议，因此需要先将其转换成一个字符串，然后依换行符分割。

这里使用了正则表达式 java.util.regex.Pattern 的一个实例 newline 来匹配换行符。尤为方便的是，Java 8 为 Pattern 类新增了一个 splitAsStream 方法，该方法使用正则表达式将字符串分割好后，生成一个包含分割结果的流对象。

用户连上聊天服务器后，首先要做的事是设置用户名。如果用户名未知，则执行设置用户名的逻辑；否则正常处理聊天消息。

还需要接收来自其他用户的消息，并且将它们传递给聊天程序客户端，让接收者能够读取消息。为了实现该功能，在设置当前用户用户名的同时，我们注册了另外一个回调，用来写入消息（例 9-3）。

例 9-3　注册聊天消息

```
eventBus.registerHandler(name, (Message<String> msg) -> {
    sendClient(msg.body());
});
```

上述代码使用了 Vert.x 的事件总线，它允许在 verticle 对象之间以非阻塞式 I/O 的方式传递消息（如图 9-1 所示）。registerHandler 方法将一个处理程序和一个地址关联，有消息发送给该地址时，就将之作为参数传递给处理程序，并且自动调用处理程序。这里使用用户名作为地址。

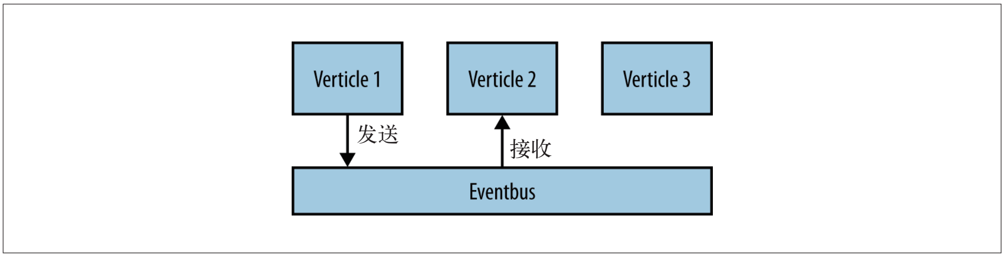


图 9-1：使用事件总线传递消息

通过为地址注册处理程序并发消息的方式，可以构建非常复杂和解耦的服务，它们之间完全以非阻塞式 I/O 方式响应。需要注意的是，在我们的设计中没有共享状态。

Vert.x 的事件总线允许发送多种类型的消息，但是它们都要使用 Message 对象进行封装。点对点的消息传递由 Message 对象本身完成，它们可能持有消息发送方的应答处理程序。在这种情况下，我们想要的是消息体，也就是文字本身，则只需调用 body 方法。我们通过将消息写入 TCP 连接，把消息发送给了用户聊天客户端。

当应用想要把消息从一个用户发送给另一个用户时，就使用代表另一个用户的地址（如例 9-4 所示），这里使用了用户的用户名。

例 9-4　发送聊天信息

```
eventBus.send(user, name.get() + '>' + message);
```

让我们扩展这个基础聊天服务器，向关注你的用户群发消息，为此，需要实现两个新命令。

- 代表群发命令的感叹号，它能将信息群发给关注你的用户。如果 Bob 键入“!hello followers”，则所有关注 Bob 的用户都会收到该条信息：“Bob>hello followers”。
- 关注命令，用来关注一个用户，比如“follow Bob”。

一旦解析了命令，就可以着手实现 broadcastMessage 和 followUser 方法，它们分别代表了这两个命令。

这里的通信模式略有不同，除了给单个用户发消息，现在还拥有了群发信息的能力。幸好，Vert.x 的事件总线允许我们将一条信息发布给多个处理程序（见图 9-2），让我们得以沿用一种类似的方式。

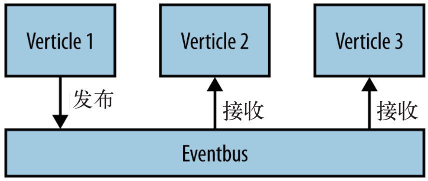


图 9-2：使用消息总线发布

代码的唯一变化是使用了事件总线的 publish 方法，而不是先前的 send 方法。为了避免用户使用！命令时和已有的地址冲突，在用户名后紧跟 .followers。比如 Bob 发布一条消息时，所有注册到 bob.followers 的处理程序都会收到消息（如例 9-5 所示）。

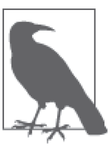
例 9-5　向关注者群发消息

```
private void broadcastMessage(String message) {
    String name = this.name.get();
    eventBus.publish(name + ".followers", name + '>' + message);
}
```

在处理程序里，我们希望和早先的操作一样：将消息传递给客户（如例 9-6 所示）。

例 9-6　接收群发的消息

```
private void followUser(String user) {
    eventBus.registerHandler(user + ".followers", (Message<String> message) -> {
        sendClient(message.body());
    });
}
```



如果将消息发送到有多个处理程序监听的地址，则会轮询决定哪个处理程序会接收到消息。这意味着在注册地址时要多加小心。

9.3　消息传递架构

这里我们要讨论的是一种基于消息传递的架构，我用它实现了一个简单的聊天客户端。聊天客户端的细节并不重要，重要的是这个模式，那就让我们来谈谈消息传递本身吧。

首先要注意的是我们的设计里不共享任何状态。verticle 对象之间通过向事件总线发送消息通信，这就是说我们不需要保护任何共享状态，因此根本不需要在代码中添加锁或使用 synchronized 关键字，编写并发程序变得更加简单。

为了确保不在 verticle 对象之间共享状态，我们对事件总线上传递的消息做了某些限

制。例子中使用的消息是普通的 Java 字符串，它们天生就是不可变的，因此可以安全地在 verticle 对象之间传递。 接收处理程序无法改变 String 对象的状态，因此不会和消息发送者互相干扰。

Vert.x 没有限制只能使用字符串传递消息，我们可以使用更复杂的 JSON 对象，甚至使用 Buffer 类构建自己的消息。这些消息是可变的，也就是说如果使用不当，消息发送者和接收者可以通过读写消息共享状态。

Vert.x 框架通过在发送消息时复制消息的方式来避免这种问题。这样既保证接收者得到了正确的结果，又不会共享状态。无论是否使用 Vert.x，确保消息不会共享状态都是最重要的。不可变消息是最简单的解决方式，但通过复制消息也能解决该问题。

使用 verticle 对象模型开发的并发系统易于测试，因为每个 verticle 对象都可以通过发送消息、验证返回值的方式单独测试。然后使用这些经过测试的模块组合成一个复杂系统，而不用担心使用共享的可变状态通信在集成时会遇到大量问题。当然，点对点的测试还是必须的，确保系统和预期的行为一致。

基于消息传递的系统让隔离错误变得简单，也便于编写可靠的代码。如果一个消息处理程序发生错误，可以选择重启本地 verticle 对象，而不用去重启整个 JVM。

在第 6 章中，我们看到了如何使用 Lambda 表达式和 Stream 类库编写并行处理数据代码。并行机制让处理海量数据的速度更快，消息传递和稍后将会介绍的响应式编程是问题的另一面：我们希望在有限的并行运行的线程里，执行更多的 I/O 操作，比如连接更多的聊天客户端。无论哪种情况，解决方案都是一样的：使用 Lambda 表达式表示行为，构建 API 来管理并发。聪明的类库意味着简单的应用代码。

9.4　末日金字塔

读者已经看到了如何使用回调和事件编写非阻塞的并发代码，但是我还没提起房间里的大象。如果编写代码时使用了大量的回调，代码会变得难于阅读，即便使用了 Lambda 表达式也是如此。让我们通过一个具体例子来更好地理解这个问题。

在编写聊天程序服务器端代码时，我写了很多测试，从客户端的角度描述了 verticle 对象的行为。代码如例 9-7 中的 messageFriend 测试所示：

例 9-7　检测聊天服务器上两个朋友是否能发消息的测试

```
@Test
public void messageFriend() {
    withModule(() -> {
        withConnection(richard -> {
            richard.dataHandler(data -> {
                assertEquals("bob>oh its you!", data.toString());
```

```
                moduleTestComplete();
            });

            richard.write("richard\n");
            withConnection(bob -> {
                bob.dataHandler(data -> {
                    assertEquals("richard>hai", data.toString());
                    bob.write("richard<oh its you!");
                });
                bob.write("bob\n");
                vertx.setTimer(6, id -> richard.write("bob<hai"));
            });
        });
    });
}
```

我连上两个客户端，分别是 Richard 和 Bob，Richard 对 Bob 说“嗨”，Bob 回答“哦，是你啊”。我已经将建立连接的通用代码重构，即使这样，读者依然会注意到那些嵌套的回调形成了一个末日金字塔。代码不断地向屏幕右方挤过去，就像一座金字塔。（别看我，这名字又不是我起的！）这是一个众所周知的反模式，让代码难于阅读和理解。同时，将代码的逻辑分散在了多个方法里。

上一章我们讨论过如何通过将一个 Lambda 表达式传给 `with` 方法的方式来管理资源。读者会注意到，在测试代码中我多次用到了该方法。`withModule` 方法部署 Vert.x 模块，运行一些代码然后关闭模块。还有一个 `withConnection` 方法连接到 `ChatVerticle`，使用完毕后关掉连接。

这里使用 `with` 方法，而不使用 `try-with-resources` 的方式，好处是它符合本章我们使用的非阻塞线程模型。我们可以重构代码，让它变得易于理解，如例 9-8 所示。

例 9-8　分成多个方法后的测试代码，测试聊天服务器上两个朋友是否能发消息

```
@Test
public void canMessageFriend() {
    withModule(this::messageFriendWithModule);
}

private void messageFriendWithModule() {
    withConnection(richard -> {
        checkBobReplies(richard);
        richard.write("richard\n");
        messageBob(richard);
    });
}

private void messageBob(NetSocket richard) {
    withConnection(messageBobWithConnection(richard));
}

private Handler<NetSocket> messageBobWithConnection(NetSocket richard) {
```

```
        return bob -> {
            checkRichardMessagedYou(bob);
            bob.write("bob\n");
            vertx.setTimer(6, id -> richard.write("bob<hai"));
        };
    }

    private void checkRichardMessagedYou(NetSocket bob) {
        bob.dataHandler(data -> {
            assertEquals("richard>hai", data.toString());
            bob.write("richard<oh its you!");
        });
    }

    private void checkBobReplies(NetSocket richard) {
        richard.dataHandler(data -> {
            assertEquals("bob>oh its you!", data.toString());
            moduleTestComplete();
        });
    }
```

例 9-8 中的重构将测试逻辑分散在了多个方法里，解决了末日金字塔问题。不再是一个方法只能有一个功能，我们将一个功能分散在了多个方法里！代码还是难于阅读，不过这次换了一个方式。

想要链接或组合的操作越多，问题就会越严重，我们需要一个更好的解决方案。

9.5 Future

构建复杂并行操作的另外一种方案是使用 Future。Future 像一张欠条，方法不是返回一个值，而是返回一个 Future 对象，该对象第一次创建时没有值，但以后能拿它“换回”一个值。

调用 Future 对象的 get 方法获取值，它会阻塞当前线程，直到返回值。可惜，和回调一样，组合 Future 对象时也有问题，我们会快速浏览这些可能碰到的问题。

我们要考虑的场景是从外部网站查找某专辑的信息。我们需要找出专辑上的曲目列表和艺术家，还要保证有足够的权限访问登录等各项服务，或者至少确保已经登录。

例 9-9 使用 Future API 解决了该问题。在❶处登录提供曲目和艺术家信息的服务，这时会返回一个 Future<Credentials> 对象，该对象包含登录信息。Future 接口支持泛型，可将 Future<Credentials> 看作是 Credentials 对象的一张欠条。

例 9-9　使用 Future 从外部网站下载专辑信息

```
    @Override
    public Album lookupByName(String albumName) {
        Future<Credentials> trackLogin = loginTo("track"); ❶
        Future<Credentials> artistLogin = loginTo("artist");
```

```
        try {
            Future<List<Track>> tracks = lookupTracks(albumName, trackLogin.get()); ❷
            Future<List<Artist>> artists = lookupArtists(albumName, artistLogin.get());

            return new Album(albumName, tracks.get(), artists.get()); ❸
        } catch (InterruptedException | ExecutionException e) {
            throw new AlbumLookupException(e.getCause()); ❹
        }
    }
```

在❷处使用登录后的凭证查询曲目和艺术家信息，通过调用 `Future` 对象的 `get` 方法获取凭证信息。在❸处构建待返回的专辑对象，这里同样调用 `get` 方法以阻塞 `Future` 对象。如果有异常，我们在❹处将其转化为一个待解问题域内的异常，然后将其抛出。

读者将会看到，如果要将 `Future` 对象的结果传给其他任务，会阻塞当前线程的执行。这会成为一个性能问题，任务不是平行执行了，而是（意外地）串行执行。

以例 9-9 来说，这意味着在登录两个服务之前，我们无法启动任何查找任务。没必要这样：`lookupTracks` 只需要自己的登录凭证，`lookupArtists` 也是一样。我们将理想的行为用图 9-3 描述出来。

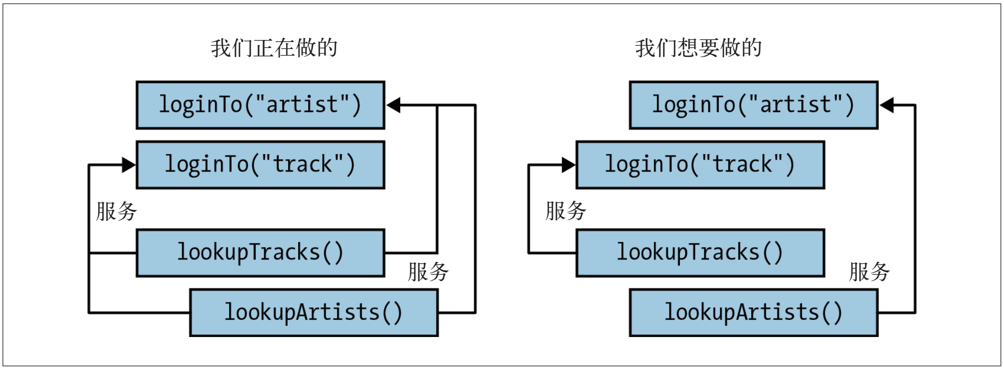


图 9-3：查询操作不必等待所有登录操作完成后才能执行

可以将对 `get` 的调用放到 `lookupTracks` 和 `lookupArtists` 方法的中间，这能解决问题，但是代码丑陋，而且无法在多次调用之间重用登录凭证。

我们真正需要的是*不必调用* `get` 方法阻塞当前线程，就能操作 `Future` 对象返回的结果。我们需要将 `Future` 和回调结合起来使用。

9.6　CompletableFuture

这些问题的解决之道是 `CompletableFuture`，它结合了 `Future` 对象打欠条的主意和使用回调处理事件驱动的任务。其要点是可以组合不同的实例，而不用担心末日金字塔问题。

> 你以前可能接触过 CompletableFuture 对象背后的概念，在其他语言中这被叫作**延迟对象**或**约定**。在 Google Guava 类库和 Spring 框架中，这被叫作 ListenableFutures。

在例 9-10 中，我会使用 CompletableFuture 重写例 9-9 来展示它的用法。

例 9-10　使用 CompletableFuture 从外部网站下载专辑信息

```
public Album lookupByName(String albumName) {
    CompletableFuture<List<Artist>> artistLookup
        = loginTo("artist")
         .thenCompose(artistLogin -> lookupArtists(albumName, artistLogin)); ❶

    return loginTo("track")
          .thenCompose(trackLogin -> lookupTracks(albumName, trackLogin)) ❷
          .thenCombine(artistLookup, (tracks, artists)
              -> new Album(albumName, tracks, artists)) ❸
          .join(); ❹
}
```

在例 9-10 中，loginTo、lookupArtists 和 lookupTracks 方法均返回 CompletableFuture，而不是 Future。CompletableFuture API 的技巧是注册 Lambda 表达式，并且把高阶函数链接起来。方法不同，但道理和 Stream API 的设计是相通的。

在❶处使用 thenCompose 方法将 Credentials 对象转换成包含艺术家信息的 CompletableFuture 对象，这就像和朋友借了点钱，然后在亚马逊上花了。你不会马上拿到新买的书——亚马逊会发给你一封电子邮件，告诉你新书正在运送途中，又是一张欠条！

在❷处还是使用了 thenCompose 方法，通过登录 Track API，将 Credentials 对象转换成包含曲目信息的 CompletableFuture 对象。这里引入了一个新方法 thenCombine ❸，该方法将一个 CompletableFuture 对象的结果和另一个 CompletableFuture 对象组合起来。组合操作是由用户提供的 Lambda 表达式完成，这里我们要使用曲目信息和艺术家信息构建一个 Album 对象。

这时我有必要提醒大家，和使用 Stream API 一样，现在还没真正开始做事呢，只是定义好了做事的规则。在调用最终的方法之前，无法保证 CompletableFuture 对象已经生成结果。CompletableFuture 对象实现了 Future 接口，可以调用 get 方法获取值。CompletableFuture 对象包含 join 方法，我们在❹处调用了该方法，它的作用和 get 方法是一样的，而且它没有使用 get 方法时令人倒胃口的检查异常。

读者现在可能已经掌握了使用 CompletableFuture 的基础，但是如何创建它们又是另外一回事。创建 CompletableFuture 对象分两部分：创建对象和传给它欠客户代码的值。

如例 9-11 所示，创建 CompletableFuture 对象非常简单，调用它的构造函数就够了。现在

就可以将该对象传给客户代码，用来将操作链接在一起。我们同时保留了对该对象的引用，以便在另一个线程里继续执行任务。

例 9-11　为 Future 提供值

```
CompletableFuture<Artist> createFuture(String id) {
    CompletableFuture<Artist> future = new CompletableFuture<>();
    startJob(future);
    return future;
}
```

一旦任务完成，不管是在哪个线程里执行的，都需要告诉 CompletableFuture 对象那个值，这份工作可以由各种线程模型完成。比如，可以 submit 一个任务给 ExecutorService，或者使用类似 Vert.x 这样基于事件循环的系统，或者直接启动一个线程来执行任务。在例 9-12 中，为了告诉 CompletableFuture 对象值已就绪，需要调用 complete 方法，是时候还债了，如图 9-4 所示。

例 9-12　为 Future 提供一个值，完成工作

```
future.complete(artist);
```

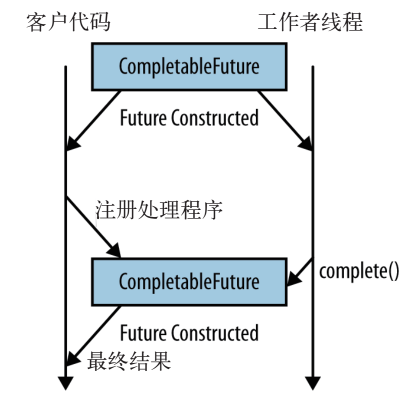


图 9-4：一个可完成的 Future 是一张可以被处理的欠条

当然，CompletableFuture 的常用情境之一是异步执行一段代码，该段代码计算并返回一个值。为了避免大家重复实现同样的代码，有一个工厂方法 supplyAsync，用来创建 CompletableFuture 实例，如例 9-13 所示。

例 9-13　异步创建 CompletableFuture 实例的示例代码

```
CompletableFuture<Track> lookupTrack(String id) {
    return CompletableFuture.supplyAsync(() -> {
        // 这里会做一些繁重的工作 ❶
        // ...
        return track; ❷
```

```
        }, service); ❸
    }
```

supplyAsync 方法接受一个 Supplier 对象作为参数，然后执行它。如❶处所示，这里的要点是能执行一些耗时的任务，同时不会阻塞当前线程——这就是方法名中 Async 的含义。❸处的返回值用来完成 CompletableFuture。在❷处我们提供了一个叫作 service 的 Executor，告诉 CompletableFuture 对象在哪里执行任务。如果没有提供 Executor，就会使用相同的 fork/join 线程池并行执行。

当然，不是所有的欠条都能兑现。有时候碰上异常，我们无力偿还，如例 9-14 所示，CompletableFuture 为此提供了 completeExceptionally，用于处理异常情况。该方法可以视作 complete 方法的备选项，但不能同时调用 complete 和 completeExceptionally 方法。

例 9-14　出现错误时完成 Future

```
    future.completeExceptionally(new AlbumLookupException("Unable to find " + name));
```

完整讨论 CompletableFuture 接口已经超出了本章的范围，很多时候它是一个隐藏大礼包。该接口有很多有用的方法，可以用你想到的任何方式组合 CompletableFuture 实例。现在，读者应该能熟练地使用高阶函数链接各种操作，告诉计算机应该做什么了吧？

让我们简单看一下其中的一些用例。

- 如果你想在链的末端执行一些代码而不返回任何值，比如 Consumer 和 Runnable，那就看看 thenAccept 和 thenRun 方法。
- 可使用 thenApply 方法转换 CompletableFuture 对象的值，有点像使用 Stream 的 map 方法。
- 在 CompletableFuture 对象出现异常时，可使用 exceptionally 方法恢复，可以将一个函数注册到该方法，返回一个替代值。
- 如果你想有一个 map，包含异常情况和正常情况，请使用 handle 方法。
- 要找出 CompletableFuture 对象到底出了什么问题，可使用 isDone 和 isCompleted-Exceptionally 方法辅助调查。

CompletableFuture 对于处理并发任务非常有用，但这并不是唯一的办法。下面要学习的概念提供了更多的灵活性，但是代码也更复杂。

9.7　响应式编程

CompletableFuture 背后的概念可以从单一的返回值推广到数据流，这就是*响应式编程*。响应式编程其实是一种声明式编程方法，它让程序员以自动流动的变化和数据流来编程。

你可以将电子表格想象成一个使用响应式编程的例子。如果在单元格 C1 中键入 =B1+5，其实是在告诉电子表格将 B1 中的值加 5，然后将结果存入 C1。而且，将来 B1 中的值变

化后，电子表格会自动刷新 C1 中的值。

RxJava 类库将这种响应式的理念移植到了 JVM。我们这里不会深入类库，只描述其中的一些关键概念。

RxJava 类库引入了一个叫作 Observable 的类，该类代表了一组待响应的事件，可以理解为一沓欠条。在 Observable 对象和第 3 章讲述的 Stream 接口之间有很强的关联。

两种情况下，都需要使用 Lambda 表达式将行为和一般的操作关联、都需要将高阶函数链接起来定义完成任务的规则。实际上，Observable 定义的很多操作都和 Stream 的相同：map、filter、reduce。

最大的不同在于用例。Stream 是为构建内存中集合的计算流程而设计的，而 RxJava 则是为了组合异步和基于事件的系统流程而设计的。它没有取数据，而是把数据放进去。换个角度理解 RxJava，它是处理一组值，而 CompletableFuture 用来处理一个值。

这次的例子是查找艺术家，如例 9-5 所示。search 方法根据名字和国籍过滤结果，它在本地缓存了一份艺术家名单，但必须从外部服务上查询艺术家信息，比如国籍。

例 9-15　通过名字和国籍查找艺术家

```
public Observable<Artist> search(String searchedName,
                                 String searchedNationality,
                                 int maxResults) {

    return getSavedArtists() ❶
          .filter(name -> name.contains(searchedName)) ❷
          .flatMap(this::lookupArtist) ❸
          .filter(artist -> artist.getNationality() ❹
                                  .contains(searchedNationality))
          .take(maxResults); ❺
}
```

在❶处取得一个包含艺术家姓名的 Observable 对象，该对象的高阶函数和 Stream 类似，在❷和❸处使用姓名和国籍做过滤，和使用 Stream 是一样的。

在❹处将姓名替换为一个 Artist 对象，如果这只是调用构造函数这么简单，我们显然会使用 map 操作。但这里我们需要组合调用一系列外部服务，每种服务都可能在它自己的线程或线程池里执行。因此，我们将名字替换为 Observable 对象，来表示一个或多个艺术家，因此使用了 flatMap 操作。

我们还需要在查找时限定返回结果的最大值：maxResults，在❺处，我们通过调用 Observable 对象的 take 方法来实现该功能。

读者会发现，这个 API 很像使用 Stream。它和 Stream 的最大区别是：Stream 是为了计算最终结果，而 RxJava 在线程模型上则像 CompletableFuture。

使用 CompletableFuture 时，我们通过给 complete 方法一个值来偿还欠条。而 Observable 代表了一个事件流，我们需要有能力传入多个值，例 9-16 展示了该怎么做。

例 9-16　给 Observable 对象传值，并且完成它

```
observer.onNext("a");
observer.onNext("b");
observer.onNext("c");
observer.onCompleted();
```

我们不停地调用 onNext 方法，Observable 对象中的每个值都调用一次。这可以在一个循环里做，也可以在任何我们想要生成值的线程里做。一旦完成了产生事件的工作，就调用 onCompleted 方法表示任务完成。和使用 Stream 一样，也有一些静态工厂方法用来从 Future、迭代器和数组中创建 Observable 对象。

和 CompletableFuture 类似，Observable 也能处理异常。如果出现错误，调用 onError 方法，如例 9-17 所示。这里的功能和 CompletableFuture 略有不同——你能得到异常发生之前所有的事件，但两种情况下，只能正常或异常地终结程序，两者只能选其一。

例 9-17　通知 Observable 对象有错误发生

```
observer.onError(new Exception());
```

和介绍 CompletableFuture 时一样，这里只给出了如何使用和在什么地方使用 Observable 的一点建议。读者如果想了解跟多细节，请阅读项目文档（https://github.com/ReactiveX/RxJava/wiki/Getting-Started）。RxJava 已经开始集成进 Java 类库的生态系统，比如企业级的集成框架 Apache Camel 已经加入了一个叫作 Camel RX（http://camel.apache.org/rx.html）的模块，该模块使得可以在该框架中使用 RxJava。Vert.x 项目也启动了一个 Rx-ify（https://github.com/vert-x/mod-rxvertx）它的 API 项目。

9.8　何时何地使用新技术

本章讲解了如何使用非阻塞式和基于事件驱动的系统。这是否意味着大家明天就要扔掉现有的 Java EE 或者 Spring 企业级 Web 应用呢？答案当然是否定的。

即使不去考虑 CompletableFuture 和 RxJava 相对较新，使用它们依然有一定的复杂度。它们用起来比到处显式使用 Future 和回调简单，但对很多问题来说，传统的阻塞式 Web 应用开发技术就足够了。如果还能用，就别修理。

当然，我也不是说阅读本章会白白浪费您一个美好的下午。事件驱动和响应式应用正在变得越来越流行，而且经常会是为你的问题建模的最好方式之一。响应式编程宣言（http://www.reactivemanifesto.org/）鼓励大家使用这种方式编写更多应用，如果它适合你的待解问题，那么就应该使用。相比阻塞式设计，有两种情况可能特别适合使用响应式或事件驱动

的方式来思考。

第一种情况是业务逻辑本身就使用事件来描述。Twitter 就是一个经典例子。Twitter 是一种订阅文字流信息的服务，用户彼此之间推送信息。使用事件驱动架构编写应用，能准确地为业务建模。图形化展示股票价格可能是另一个例子，每一次价格的变动都可认为是一个事件。

另一种显然的用例是应用需要同时处理大量 I/O 操作。阻塞式 I/O 需要同时使用大量线程，这会导致大量锁之间的竞争和太多的上下文切换。如果想要处理成千上万的连接，非阻塞式 I/O 通常是更好的选择。

9.9　要点回顾

- 使用基于 Lambda 表达式的回调，很容易实现事件驱动架构。
- `CompletableFuture` 代表了 IOU，使用 Lambda 表达式能方便地组合、合并。
- `Observable` 继承了 `CompletableFuture` 的概念，用来处理数据流。

9.10　练习

本章只有一个练习：使用 `CompletableFuture` 重构代码。先以例 9-18 中所示的 `BlockingArtistAnalyzer` 类开始，该类从两个艺术家的名字中找出成员数更多的那个，如果第一个艺术家的成员多，返回 `true`，否则返回 `false`。该类被注入一个 `artistLookupService`，因为查找 `Artist` 的过程可能会耗费一定时间。由于 `BlockingArtistAnalyzer` 类要依序调用两次查找服务，分析就会变慢，练习的目标就是加速这一过程。

例 9-18　`BlockingArtistAnalyzer` 告诉用户哪位艺术家的成员更多

```java
public class BlockingArtistAnalyzer {

    private final Function<String, Artist> artistLookupService;

    public BlockingArtistAnalyzer(Function<String, Artist> artistLookupService) {
        this.artistLookupService = artistLookupService;
    }

    public boolean isLargerGroup(String artistName, String otherArtistName) {
        return getNumberOfMembers(artistName) > getNumberOfMembers(otherArtistName);
    }

    private long getNumberOfMembers(String artistName) {
        return artistLookupService.apply(artistName)
                                  .getMembers()
                                  .count();
    }
}
```

练习分成两部分，第一部分是使用一个回调接口重构阻塞代码。在这里，我们将使用 Consumer<Boolean>，Consumer 是 JVM 自带的一个函数接口，接受一个参数，返回空。读者的任务就是修改 BlockingArtistAnalyzer，实现 ArtistAnalyzer（如例 9-19 所示）。

例 9-19　需要实现的 ArtistAnalyzer 接口

```
public interface ArtistAnalyzer {

    public void isLargerGroup(String artistName,
                              String otherArtistName,
                              Consumer<Boolean> handler);
}
```

现在我们有了一个符合回调模型的 API，就不需要同时执行两次阻塞式的查找了。使用 CompletableFuture 类重构 isLargerGroup 方法，让其可以并行执行。

第10章

下一步该怎么办

Java 作为一门语言，在很多方面都经受住了时间的考验。它仍然是非常受欢迎的平台，选用 Java 开发企业级应用是个不错的选择。人们开发了大量的开源类库和框架，解决各种各样的问题：从编写模块化且复杂的网络应用（Spring 框架）到正确地计算日期和时间（Jodatime 类库）。开发工具更是无可比拟，集成开发环境有 Eclipse 和 Intellij，构建系统有 Gradle 和 Maven。

问题在于，多年来，Java 没有紧跟时代向前演进，落得个保守的坏名声。之所以如此，部分原因也在于它流行的时间太长；亲不尊，熟生蔑，它太为人所熟悉反而容易被轻慢。当然，Java 的发展也的确存在问题。保持向后兼容的决策，尽管有所裨益，却太过复杂。

所幸，Java 8 的出现是一个积极的信号，它不仅是对语言本身的一小步改善，也是在 Java 开发方面迈出的一大步。和 Java 6、Java 7 不同，Java 8 不再是一些无足轻重的对类库的改良，以后的版本也该沿袭 Java 8 的传统，大踏步前进。不仅因为我喜欢写这一主题的书，也因为在提高编程的基本任务方面还有很长的路要走：如何把程序写得易读？如何明确地表明程序的意图？如何让高性能程序易于编写？唯一的遗憾在于这概括性的一章篇幅太短，很难完整描述出后续版本的潜力。

本书已接近尾声，但希望读者学习和使用 Java 8 的脚步不会停留在此。本书描述了各种使用 Lambda 表达式的方式：更好的集合类代码、数据并行处理、更简洁干净的代码、并发。书中阐释了为什么使用 Lambda 表达式、Lambda 表达式是什么，以及怎么用 Lambda 表达式，但一切还在于读者如何真正将其应用于实践。本着这种精神，这里给出一些开放性的练习，没有标准答案，理解这些问题能够指导读者接下来的学习过程。

- 向其他程序员（朋友或同事）解释什么是 Lambda 表达式，为什么会对它产生兴趣。
- 尝试将目前从事的项目部署到 Java 8 环境下。如果现有单元测试已经能运行在持续集成系统 Jenkins 下，那么在多个版本的 Java 上构建程序也易如反掌。
- 使用新的 Stream 和 Collector，开始重构真实产品中的遗留代码。它既可以是感兴趣的开放源码项目，也可以是当前从事的项目，前提是第一步里已经部署成功一个测试环境。如果还没准备好大规模迁往 Java 8，那么在分支上使用 Java 8 做一些原型会是个不错的开始。
- 有没有一些大规模处理数据的代码？或者代码中存在并发问题？试着使用 Stream 处理数据，或使用 RxJava 中新的并发特性，也可以使用 `CompletableFuture` 类，来重构你的代码。

选择一个熟悉的代码库，分析它的设计和架构。

- 从宏观上看，有没有更好的实现方法？
- 能否简化设计？
- 能否减少实现某功能所需的代码量？
- 怎样让代码更易读？

封面介绍

本书封面上的动物是小乌雕（Aquila pomarina），这种体型较大的猛禽分布于东欧，和其他常见的鹰一样，它也属于鹰科。小乌雕体型中等，头和喙比鹰的小，一般身长 60 厘米，翼展 150 厘米。

未成年的小乌雕飞羽上有白色斑点，而成年后，头上的羽毛呈现浅褐色，羽翼则变为深黑色。小乌雕主要在中欧和东欧地区繁衍，它们在树上筑巢，每次产 1 至 3 个有米黄色斑点的蛋。和所有的鹰一样，幼鸟的数量取决于繁殖季节捕食的数量。雌鸟产下第一枚蛋之后就开始孵蛋，第一个破壳而出的幼鸟常常会破坏或吃掉其他鸟蛋。

封面图片由 Meyers Kleines 提供。

O'REILLY®

Java 8函数式编程

对于有经验的Java程序员来说，全面了解Java 8引入的Lambda表达式是当务之急。本书作者是资深Java开发者、英国伦敦Java社区负责人，英文原版深受好评，被誉为学习Lambda表达式的必读佳作。这本书言简意赅，示例精到，全面介绍了因为Lambda表达式的引入，Java这门世界上最流行的语言都发生了哪些重大变化，以及匿名函数将如何重塑Java的编程范式。全书篇幅不长，环环相扣，读来令人手不释卷。

函数式编程的确能大幅提升编程效率，但它也并不高深，绝非少数人的游戏。本书可以让所有Java程序员平滑过渡到Java 8时代。前半部分展示了如何正确使用Lambda表达式；后面几章介绍如何利用Lambda表达式提高并发操作的性能、编写出更简单的并发代码。全书采用了示例驱动的写作风格：每介绍完一个概念，紧接着给出一段示例代码，并辅以详尽的讲解。多数章节还在最后提供了练习题，供读者自行练习。

本书主要内容：

- 通过每一章的练习快速掌握Java 8中的Lambda表达式
- 分析流、高级集合和其他Java 8类库的改进
- 利用多核CPU提高数据并发的性能
- 将现有代码库和库代码Lambda化
- 学习Lambda表达式单元测试和调试的实践解决方案
- 用Lambda表达式实现面向对象编程的SOLID原则
- 编写能有效执行消息传送和非阻塞I/O的并发应用

Richard Warburton 一位经验丰富的技术专家，善于解决复杂深奥的技术问题，拥有华威大学计算机科学专业博士学位。近期他一直从事高性能计算方面的数据分析工作。他是英国伦敦Java社区的领导者，组织过面向Java 8中Lambda表达式、日期和时间的Adopt-a-JSR项目，以及Openjdk Hackdays活动。Richard还是知名的会议演讲嘉宾，曾在JavaOne、DevoxxUK和JAX London等会议上演讲。

“本书最出色的地方在于，它脉络清晰地说明了为什么、在何处以及如何使用Lambda表达式，激励读者改善自己的代码库。”

——**Martijn Verburg**
jClarity公司CEO，Java Champion

“我超级推荐本书，每个想了解JDK 8新特性的开发人员都应该人手一本。”

——**Daniel Bryant**
Instant Access技术公司CTO

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/程序设计/Java

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-38488-1

ISBN 978-7-115-38488-1
定价：39.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks